[美]Ray Barrera等 著 张颖 译

Unity人工智能游戏开发（第2版）

Unity AI Game Programming,
Second Edition

清华大学出版社

Unity AI Game Programming,Second Edition

清华大学出版社数字出版网站
WQBook 文泉局
www.wqbook.com

ISBN 978-7-302-44690-3
9 787302 446903 >
定价：49.00元

Unity 人工智能游戏开发（第 2 版）

[美] Ray Barrera 等著

张 颖 译

清华大学出版社

北 京

内 容 简 介

本书详细阐述了与 Unity 游戏人工智能相关的基本解决方案，主要包括游戏 AI 的基础知识、有限状态机、实现感知系统、寻路方案、群集行为、行为树、模糊逻辑等内容。此外，本书还提供了相应的示例、代码，以帮助读者进一步理解相关方案的实现过程。

本书适合作为高等院校计算机及相关专业的教材和教学参考书，也可作为相关开发人员的自学教材和参考手册。

北京市版权局著作权合同登记号 图字：01-2016-5195

图书在版编目（CIP）数据

Unity 人工智能游戏开发：第 2 版/（美）雷・巴雷拉（Ray Barrera）等著；张颖译. —北京：清华大学出版社，2016
书名原文：Unity AI Game Programming-Second Edition
ISBN 978-7-302-44690-3

I. ①U…　II. ①雷…　②张…　III. ①游戏程序-程序设计　IV. ①TP311.5

中国版本图书馆 CIP 数据核字（2016）第 184628 号

责任编辑：贾小红
封面设计：刘　超
版式设计：魏　远
责任校对：王　云
责任印制：何　芊

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 刷 者：三河市君旺印务有限公司
装 订 者：三河市新茂装订有限公司
经　　销：全国新华书店
开　　本：185mm×230mm　　**印　　张**：12　　**字　　数**：226 千字
版　　次：2016 年 9 月第 1 版　　**印　　次**：2016 年 9 月第1 次印刷
印　　数：1～3000
定　　价：49.00 元

产品编号：068954-01

译 者 序

Unity 是近几年非常流行的一个 3D 游戏开发引擎（特别是移动平台），它的特点是跨平台能力强，支持 PC、Mac、Linux、网页、iOS、Android 等几乎所有的平台，移植便捷，3D 图形性能出众，为众多游戏开发者所喜爱。在手机平台，Unity 几乎成为 3D 游戏开发的标准工具。

Unity 向开发人员提供了多种工具，以实现具有人工智能的游戏体验。Unity 的内建 API 以及特性可有效地实现多种可能性，并构建游戏场景和角色对象。无论开发何种游戏，理解并应用人工智能特性可视为游戏设计的基本因素之一。本书将人工智能技术划分为多个简单概念，进而有助于读者理解这一话题的基础内容。本书通过大量实例，详细分析所涉及的概念，并对核心概念予以实现。

在此基础上，本书还将引领读者实现自己的状态机模式，其中涉及了 AI 的基础型传感器系统，并将其整合至有限状态机中。随后，读者还将领略 Unity 内建的 NavMesh 特性，并实现自己的 A*寻路系统。本书还讲解了群集行为这一核心 AI 概念的实现，并了解行为树的工作和实现方式。最后，状态机系统中还将进一步加入模糊逻辑，并最终实现一个完整的小型游戏项目。

本书将 Unity 与人工智能进行有机的结合，并讨论较为高级的开发技术和解决方案。

在本书的翻译过程中，除张颖外，程聪、解宝香、景秀红、李保金、李莉、李亚楠、梁洪娇、林芮、刘鹤等人也参与了部分翻译工作，在此一并表示感谢。

限于译者的水平，译文中难免有错误和不妥之处，恳请广大读者批评指正。

译　者

前　言

本书讨论与游戏开发相关的人工智能（AI）技术，并展示了游戏开发过程中的多种应用，或许其处理方式超出了我们的想象。

如果读者想成为 AI 领域的专家，须经历漫长的学习过程。本书提供了与 AI 相关的技术知识和开发工具，读者可在自己的游戏项目中实现 AI，并在此基础上对其进行扩展或创新。

本书示例均配备了相应的代码和项目文件，各章涵盖了与相关概念相符的背景知识以及示例。读者理解后可将其应用于自己的开发项目中。

本书内容

考虑 AI 的广泛性和重要性，第 1 章阐述了与其相关的基本概念。

第 2 章探讨 AI 中应用十分广泛的概念，即有限状态机。

第 3 章介绍游戏 AI 主体的重要实现方式，进而对周边场景予以感知。AI 主体的真实性直接关系到与周围环境间的响应方式。

第 4 章讨论游戏 AI 主体中的路径搜索模式。游戏中的 AI 需要遍历游戏关卡区域，并躲避途中的障碍物。

第 5 章介绍群集模拟算法，以使读者可处理游戏中主体间的整体运动，而非各独立体的单一逻辑。

第 6 章实现了自定义行为树，这也是游戏中复杂 AI 行为中较为常见的处理方式。

第 7 章根据多种因素探讨了游戏 AI 主体的决策制定方式。其中，模糊逻辑可用于模拟人类的决策方式。

第 8 章讨论单目标游戏模板中各类系统间的整合方式，并可方便地对其进行扩展。

背景知识

当运行本书提供的示例项目时，读者需要获取一份 Unity 5 副本。对此，可访问 https://unity3d.com/get-unity 并免费下载 Unity 5 的最新版本。另外，关于 Unity 的系统需求条件，读者可访问 https://unity3d.com/get-unity。

Unity 5 绑定了 MonoDevelop，但本书并不推荐使用该软件，常见的 IDE 即可满足本

书的文本编辑要求。然而，MonoDevelop 涵盖了编写、调试代码的一切内容，包括自动补齐功能，且无须使用任何插件和扩展。

适用读者

本书假定读者已基本了解 C#语言以及 Unity 编辑器，但希望能够编写首款属于自己的游戏，并扩展个人的游戏开发知识。本书提供了大量的与游戏 AI 相关的示例，且不要求读者具备与游戏 AI 相关的任何技术背景知识。

本书约定

本书涵盖了多种文本风格，进而对不同类型的信息加以区分。下列内容展示了对应示例及其具体含义。

文本中的代码、数据库表名称、文件夹名称、文件名、文件扩展名、路径名、伪 URL、用户输入以及推特用户名采用如下方式表示：

" We'll name it TankFsm"。

代码块则通过下列方式设置：

```
using UnityEngine;
using System.Collections;

public class TankPatrolState : StateMachineBehaviour {

  //OnStateEnter is called when a transition starts and the state
machine starts to evaluate this state
  //override public void OnStateEnter(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex) {
  //
  //}

  //OnStateUpdate is called on each Update frame between
OnStateEnter and OnStateExit callbacks
  //override public void OnStateUpdate(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex) {
  //
```

```
  //}

  //OnStateExit is called when a transition ends and the state
machine finishes evaluating this state
  //override public void OnStateExit(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex) {
  //
  //}

  //OnStateMove is called right after Animator.OnAnimatorMove().
Code that processes and affects root motion should be implemented here
  //override public void OnStateMove(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex) {
  //
  //}

  //OnStateIK is called right after Animator.OnAnimatorIK(). Code
that sets up animation IK (inverse kinematics) should be implemented
here.
  //override public void OnStateIK(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex) {
  //
  //}
}
```

最后，“提示”表示一些较为重要的提示；“技巧”则表示相关的操作技巧。

读者反馈和客户支持

欢迎读者对本书的建议或意见予以反馈，以进一步了解读者的阅读喜好。反馈意见对于我们来说十分重要，以便改进我们日后的工作。

对此，读者可向 feedback@packtpub.com 发送邮件，并以书名作为邮件标题。

若读者针对某项技术具有专家级的见解，抑或计划撰写书籍或完善某部著作的出版工作，则可阅读 www.packtpub.com/authors 中的 author guide 一栏。

作为本书的读者支持，我们将对每一名用户提供竭诚的服务。

资源下载

读者可访问 http://www.packtpub.com 下载本书中的示例代码文件；或者访问 http://www.packtpub.com/support，经注册后可直接通过邮件方式获取相关文件。

另外，我们还以 PDF 文件方式提供了本书中截图、图表的彩色图像，以帮助读者进一步理解输出结果中的变化，读者可访问 https://www.packtpub.com/sites/default/files/downloads/8272OT_ColorImages.pdf 下载该 PDF 文件。

勘误表

尽管我们在最大程度上做到尽善尽美，但错误依然在所难免。如果读者发现谬误之处，无论是文字错误抑或是代码错误，还望不吝赐教。对于其他读者以及本书的再版工作，这将具有十分重要的意义。对此，读者可访问 http://www.packtpub.com/submit-errata，选取对应书籍，单击 ErrataSubmissionForm 超链接，并输入相关问题的详细内容。经确认后，填写内容将被提交至网站，或添加至现有勘误表中（位于该书籍的 Errata 部分）。

另外，读者还可访问 http://www.packtpub.com/books/content/support 查看之前的勘误表。在搜索框中输入书名后，所需信息将显示于 Errata 项中。

版权须知

一直以来，互联网上的版权问题从未间断，Packt 出版社对此类问题异常重视。若读者在互联网上发现本书任意形式的副本，请告知网络地址或网站名称，我们将对此予以处理。

关于盗版问题，读者可发送邮件至 copyright@packtpub.com。

对于作者的爱护，我们表示衷心的感谢，并于日后向读者呈现更为精彩的作品。

问题解答

若读者对本书有任何疑问，均可发送邮件至 questions@packtpub.com，我们将竭诚为您服务。

目　录

第 1 章　游戏 AI 的基础知识

一般来讲，人工智能（AI）是一个庞杂的话题，其内容较为艰深，但应用范围却十分规范，例如机器人学、统计学、娱乐业（近期展示出了强大的势头）以及视频游戏。本章目标即是将 AI 应用划分为多种彼此关联同时切实可行的方案，对应示例进一步阐述了相关概念，并直奔主题。

本章还将介绍某些在学术领域、传统领域和游戏领域内与 AI 相关的背景知识，其中包括：

- AI 在游戏中的应用与实现有别于其他领域。
- 考察 AI 在游戏中的特定需求条件。
- 考察游戏中基本的 AI 模式。

本章讨论了 Unity 中 AI 的实现模式，并可作为其他章节的参考内容。

1.1　创建生活幻象

生物体通常包含一定的智能，例如动物和人类，进而可制定某种执行决策。人类的大脑可通过声音、触觉、气味或者视觉对刺激行为予以反馈，并于随后将此类数据转换为可处理的信息。另外一方面，作为一类电子设备，计算机可接收二进制信息，高速执行逻辑和数学计算，并输出最终结果。因此，AI 实际上使得计算机设备具有某种思考能力，并像有机生物一样执行特定操作。

AI 及其相关知识涵盖了大量的内容，在深入讨论这一主题之前，读者应理解 AI 在不同领域内的基本应用。作为一类通用术语，AI 的实现和应用针对不同领域具有不同的处理方式，进而求解不同的问题集。

在考察特定的游戏技术之前，下面首先讨论某些与 AI 相关的科研领域，这些领域在近些年来取得了重大的进步。某些在科幻小说中出现的场景现已成为科学事实，例如自主机器人。AI 的发展也体现在日常的生活中，例如，智能手机即包含了数字辅助特征，并与某些最新出现的 AI 技术有关。下列科研领域对 AI 技术的发展均起到了

推动作用。

- 计算机视觉：从视频和摄像头中获取输入，对其进行分析后执行特定的操作，例如面部识别、物体识别以及光电字符识别。
- 自然语言处理（NLP）：机器能够像人类那样正常地读写并理解自然语言。该问题可描述为，机器通常难以理解人类语言，相同事物存在不同的阐述方式；根据上下文，同一句子也包含了不同的含义。在处理并做出响应之前，由于需要理解人类的语言和表达方式，因而 NLP 是机器处理过程中的一个重要步骤。对此，Web 中存在大量的数据集，以帮助科研工作者进行语言的自动分析处理。
- 常识推理：即使在不熟悉的领域，人类的大脑也可从中获取某种答案。鉴于大脑可在上下文、背景知识以及语言能力间混合、交互信息，因而常识性知识对于人类而言并不是问题。然而，机器处理过程则异常复杂，对于科研工作者而言依然是一项艰巨的挑战。
- 机器学习：这一话题听起来像是取材于科幻电影，但现实与科幻间的差距并不遥远。计算机程序一般包含静态指令集，接受输入数据并提供输出结果。机器学习强调算法和程序，并可从程序所处理的数据中进行学习。

1.2　利用 AI 进一步完善游戏

游戏中的 AI 可追溯到早期作品，例如 Namco 发布的“吃豆人”游戏，其 AI 尚处于初级阶段。但即使如此，敌方角色（Blinky、Pinky、Inky 和 Clyde）均具有各自的行为模式，并通过多种方式向玩家发起挑战。游戏难度的增加使得玩家乐此不疲，自其发布 30 多年来，玩家们依然对此津津乐道。

游戏设计师负责掌控游戏的难度，并制定游戏与玩家的平衡性。最终，AI 可视为一类强大的工具，并有助于实现游戏实体行为模式的抽象化操作，进而使其更具真实感。类似于动画设计师进行逐帧设计，或者艺术家利用手中的画笔进行艺术创作，设计人员或程序员同样可发挥其创造力，并巧妙地利用 AI 技术对游戏予以完善。

AI 在游戏中扮演的角色旨在增加游戏的趣味性，即提供具有一定挑战性的内容，以及相对有趣的非玩家控制角色（即 NPC），并在游戏场景中呈现较为真实的运动行为。当前目标并非是复制人类或动物的全部思考过程，仅是呈现某种幻象以使 NPC 具

有一定的智能，并通过玩家眼中相对正确的方式与游戏场景中变化的环境进行互动。

尽管技术上可设计、构建复杂的模式和行为，但游戏 AI 尚不能模拟真实的人类行为。功能强大的微芯片、大容量内存，甚至是分布式计算，大大提升了 AI 的计算能力。总体上看，数据资源依然会在其他操作间被共享，例如图形渲染、物理模拟、音频处理以及动画等，全部内容均以实时方式呈现。同时，系统间须实现完美整合，并在游戏中提供稳定的帧速率。与游戏开发过程中的其他准则类似，优化 AI 计算对于开发人员而言仍然颇具挑战性。

1.3　在 Unity 使用 AI

本节引领读者大致浏览不同游戏类型中所用的 AI 技术，后续章节将逐一展现 Unity 中的各项特性。Unity 引擎具有强大的灵活性，并提供了多种 AI 模式的实现方法。其中，某些方法可直接使用，而另一些方案则需要从头开始加以构建。本书主要讨论 Unity 中基础的 AI 模式，以使游戏中的 AI 实体正常运行。AI 领域涵盖了大量的内容，本书仅是万里长征的第一步。

1.4　定 义 主 体

在讨论具体技术之前，首先需要明晰本书所采用的一个关键术语，即主体(agent)。考虑到与 AI 紧密相关，因而主体表示为具有人工智能的实体。当谈及 AI 时，并非特指某一具体角色，而是指呈现复杂行为模式的某一实体，且具有非随机性或智能特征。此类实体可以是一个角色、生物体、车辆或者其他事物。主体定义为一类自治实体，并执行所制定的模式和行为。后续章节将据此展开讨论。

1.5　有限状态机概述

有限状态机（FSM）可视为最为简单的 AI 模型之一，且常出现于游戏中。状态机通常包含了一组状态，并通过图中的转换予以连接。一般情况下，游戏实体始于某一初始状态，并搜索触发状态间转换的事件和规则。在任意既定时刻，游戏实体仅处

于一种状态之下。

例如，在典型的射击类游戏中，AI 防御角色通常具有巡逻模式、追逐模式和射击模式，如图 1.1 所示。

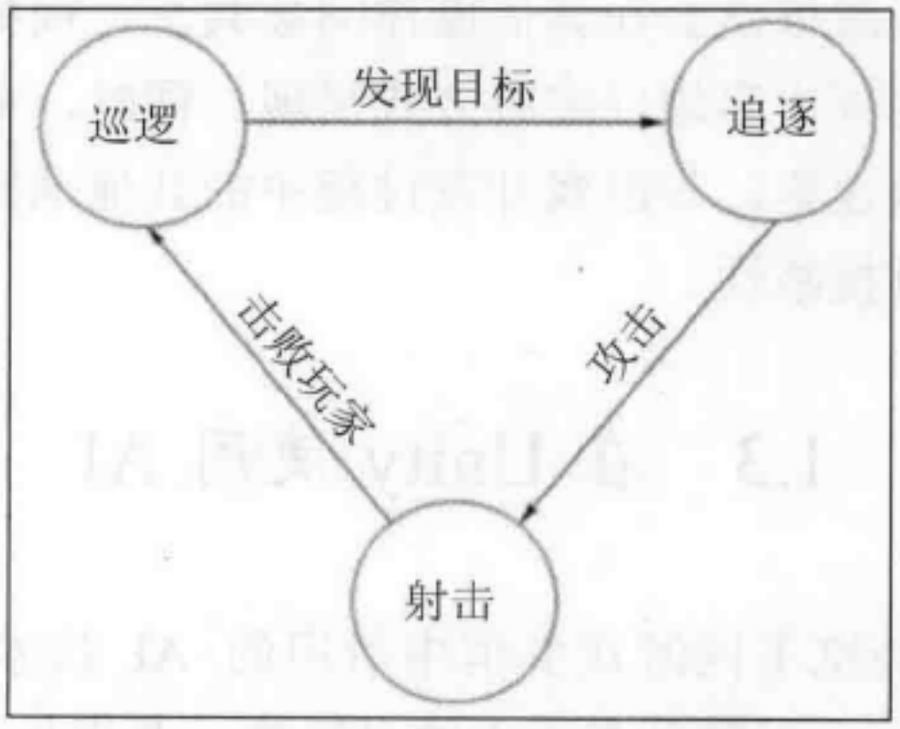

图 1.1

简单的 FSM 中一般包含下列 4 个组件。

- 状态：该组件定义了一组游戏实体或 NPC 可选择的一组唯一状态（例如巡逻、追逐和射击）。
- 转换：该组件定义了不同状态间的关系。
- 规则：该组件用于触发某一状态转换（例如玩家巡视、位于射杀范围以及玩家被摧毁）。
- 事件：该组件负责触发规则检测操作（守卫的可见区域、与玩家间的距离等）。

鉴于实现的简单性、可视化特征以及易于理解，FSM 常用于游戏开发过程中的搜寻 AI 模式。对此，使用简单的 if/else 语句或 switch 语句，即可实现 FSM。如果开始阶段即包含诸多状态和转换，则事态将会变得较为复杂。第 2 章将对 FSM 的管理方式加以深入讨论。

1.6　通过主体视角查看场景

为了保证 AI 行为确实可信，主体对象需要与其周边事件、环境、玩家，甚至是其他主体进行反馈。类似于真实的有机生物，此类主体依赖于视线、声音以及其他物理“刺激”。然而，与真实生物体感知其周围环境相比，主体对象可访问游戏中的大

量数据，例如玩家的位置、距离、库存量、道具的位置，以及在代码中暴露于该主体对象的任意变量。

在图 1.2 中，当前主体对象的视野通过其前方锥体表示，而听觉范围则采用周围的灰色圆形表示。

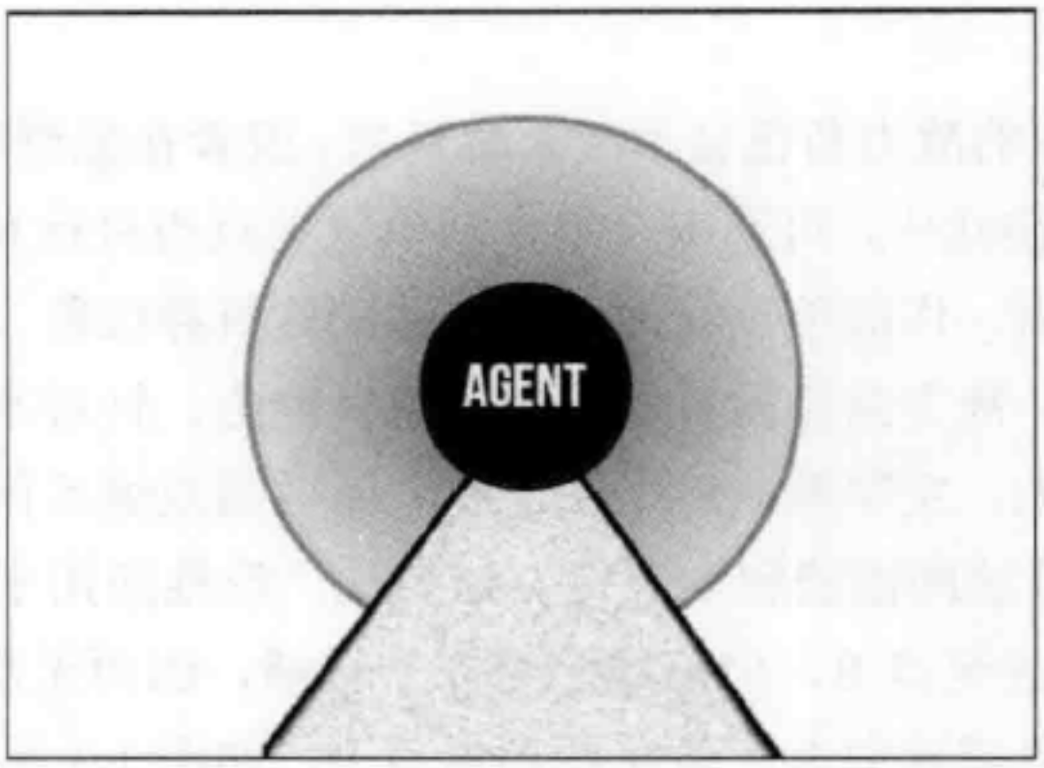

图 1.2

基本上讲，视觉、声音以及其他感觉均可视为数据。其中，视觉表示为光线粒子，声音则表示为一种振动方式等。当然，本书并不会讨论光线粒子反射过程中的复杂行为及其与主体间的作用方式，而是通过某种方式对数据进行建模，进而获得类似的结果。

可以想象，还可采用类似方式对其他感知系统进行建模，其范围并不仅限于生物体（包括视觉、声音或气味），甚至还可以是数字和机械系统，例如敌方的机器人、炮塔、声呐或雷达系统。

1.7　路 径 跟 踪

某些时候，AI 角色需要在游戏场景中按照大致路线或既定路线行进。例如，在赛车类游戏中，AI 车辆需要在公路上进行导航；在 RTS 游戏中，其他作战单位须知晓玩家发出的作战位置，并实现整体前进。

为了进一步展示智能特征，主体对象需要确定目标位置，判断是否可到达该点，选取最佳路线，并在遇到障碍物时调整路线。在后续章节中将会讨论到，各条路径均

可采用有限状态机描述，并查看此类系统间的关联方式。

本书将介绍基本的寻路和导航方法，首先讨论 A*寻路系统，而随后阐述 Unity 内置的导航网格（NavMesh）特性。

1.7.1　A*寻路

不难发现，游戏中的敌方角色会不断追逐玩家，或者在躲避障碍物时到达特定点。例如，在典型的 RTS 游戏中，可选取一组作战单位并点击目标地点，或点击敌方角色以对其进行攻击。随后，作战单位将搜寻路径并到达目标位置，期间不会与任何障碍物发生碰撞。相应地，敌方角色同样需要具备这种能力。针对不同的作战单位，障碍物也将发生变化。例如，空军编队将飞越山脉，而地面或炮兵部队则需要搜索相关行进路线。考虑到 A*算法的性能和准确性，该算法广泛地应用于游戏中。假设当前作战单位需要从点 A 行进至点 B，但路线上存在一堵墙，因而无法直接抵达目标位置。因此，应避开墙体，且需要获取一条路线到达点 B，如图 1.3 所示。

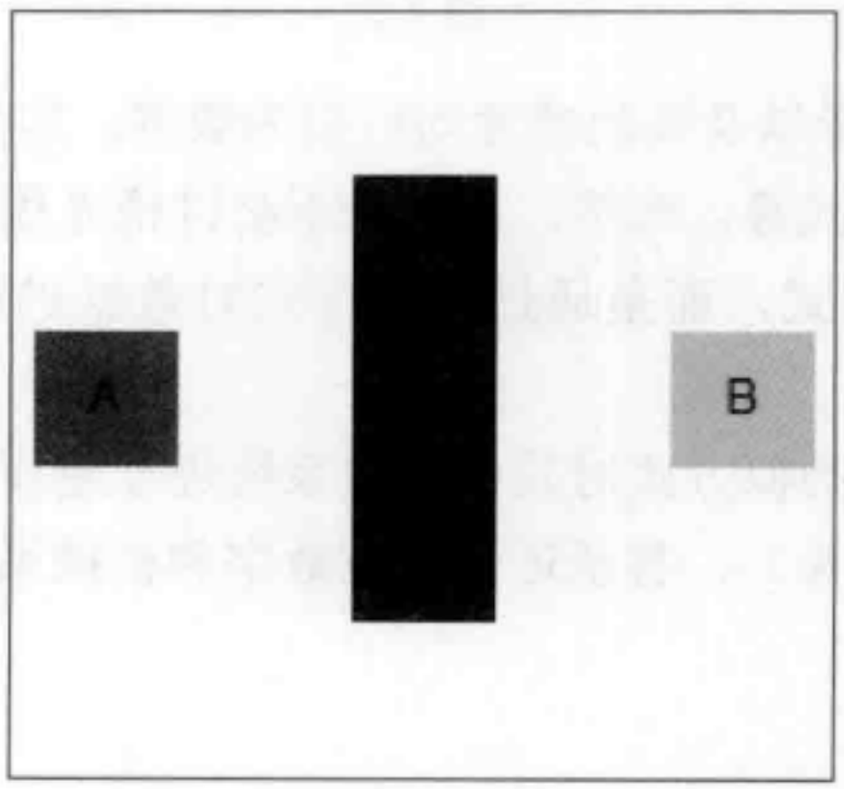

图 1.3

为了获取点 A 至点 B 间的一条路径，需要对地图进行分析，例如障碍物的位置。对此，需要将地图划分为较小的单元，并通过网格形式描述全部地图。其中，单元可表示为多种形状，例如六边形或三角形。采用网格方式描述地图可简化搜索区域，这在路径搜索机制中是一个较为重要的步骤。图 1.4 采用 2D 网格阵列表示地图。

当地图通过一系列的单元格表示后，可搜索最佳路径并到达目标位置，即计算与起始单元格邻接的各个单元格（未被障碍物占据）的运动值，并于随后选择包含最低

估值的单元格，如图 1.5 所示。第 4 章将对此加以简要介绍。

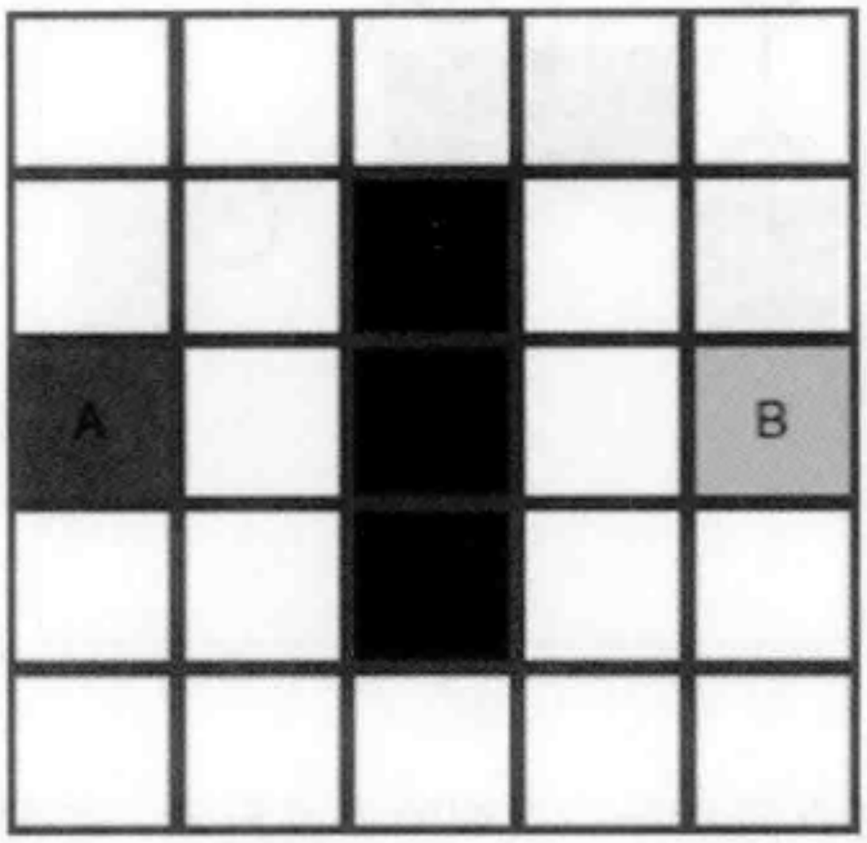

图 1.4

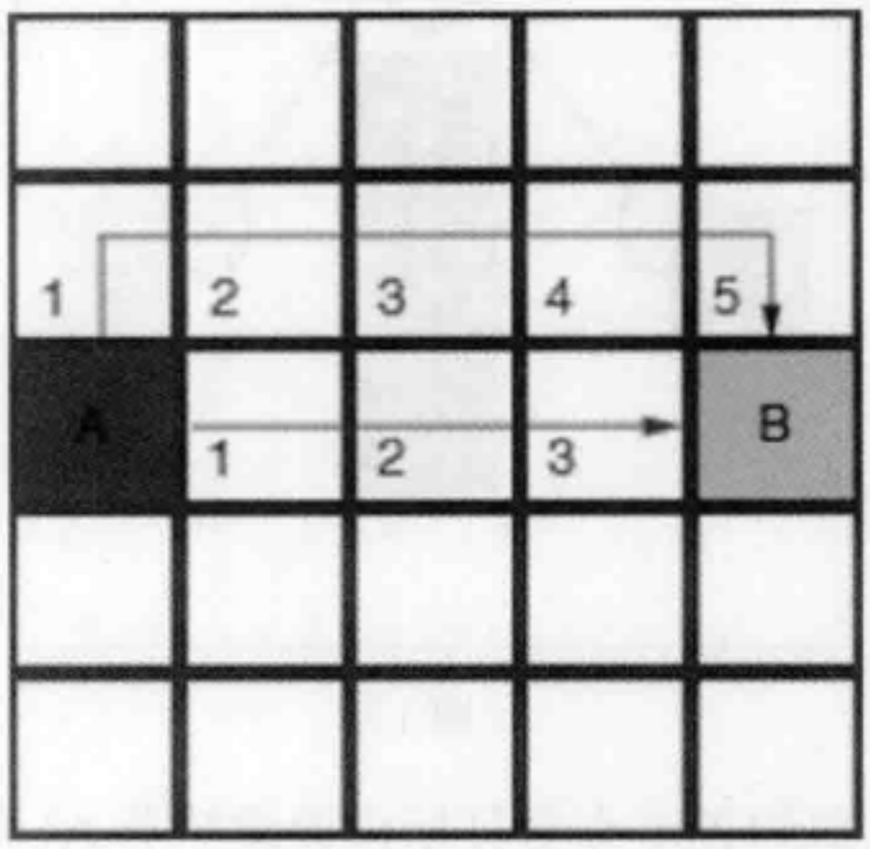

图 1.5

A*算法是一类较为重要的路径搜索方法，Unity 也提供了内置的相关特性，例如自动导航网格，以及 NavMesh 主体对象，第 4 章将对此加以讨论。无论使用 A*算法或 Unity 内置的 NavMesh 方案，具体操作均与实际项目的需求条件相关，且每种方案均包含自身的优势和缺陷。因此，全面了解两种方法有助于读者选取最佳处理方案。下面将对 NavMesh 加以介绍。

1.7.2　使用网格导航

1.7.1 节简要介绍了 A*算法，本节将通过 NavMesh 了解网格计算的适配方案。需要注意的是，为了获得相对于目标位置的最短路径（同时躲避障碍物），在 A*算法中采用简单的网格依然会占用大量的计算。因此，为了获得简单有效的 AI 角色路径搜索方法，可采用路点作为导向，并在点 A 和点 B 之间移动 AI 角色。图 1.6 中包含了 3 个路点。

当前任务是拾取最近路点，并跟随其连接节点到达目标路点。鉴于简单、高效以及较少的计算量，因而多数游戏采用了路点寻路。尽管如此，路点法依然包含了某些问题。例如，若需要更新地图中的障碍物，情况又当如何？对于地图的更新行为，需要再次设置路点，如图 1.7 所示。

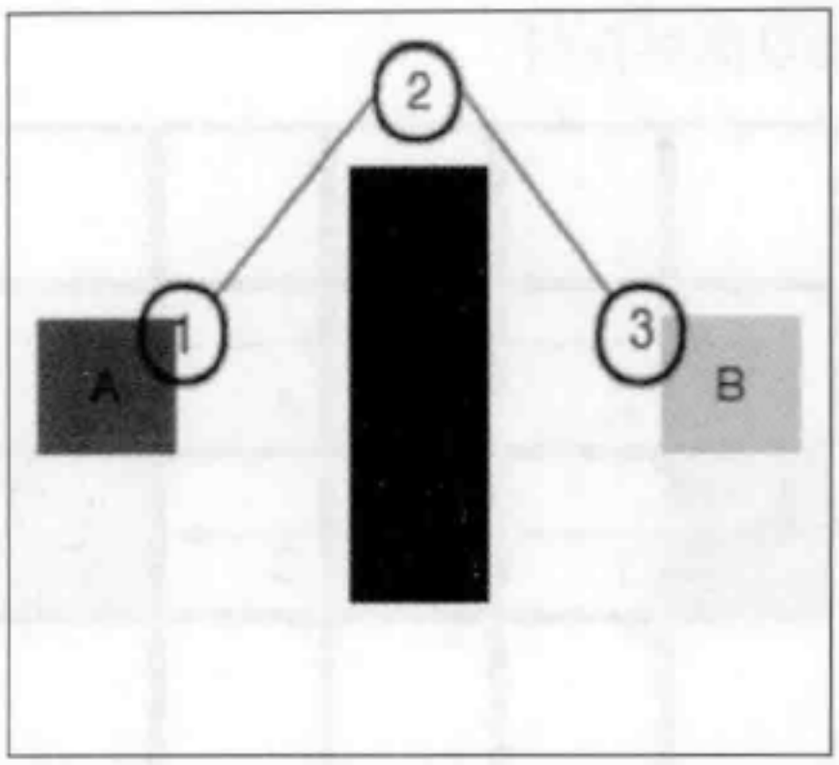

图 1.6

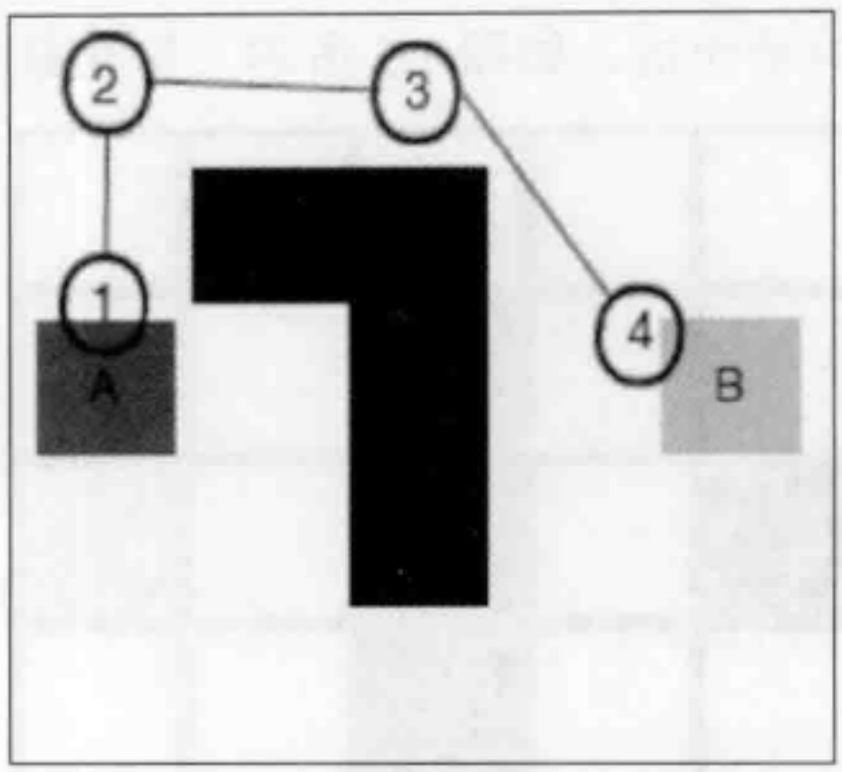

图 1.7

跟随各节点至目标位置意味着 AI 角色在节点间执行一系列的直线运动，在图 1.7 中，AI 角色有可能在近墙体路径上与其发生碰撞。对此，AI 可尝试穿越墙体到达下一个目标位置，以避免造成阻塞。即使转换至样条以使路径呈现为更加平滑的状态，并进行适当调整以躲避障碍物，路点依然无法生成与当前环境相关的信息，尽管节点之间采用样条加以连接。当平滑且经过调整后的路径途径悬崖边缘或桥边时，情况又如何？对应路径可能不再安全。因此，当 AI 实体应在全部关卡中行进时，需要使用到大量的路点，这将极大地增加实现和管理的难度。

NavMesh 是另一种图形结构，并可用于表现场景内容，该结构与正方形单元网格或路点图类似，如图 1.8 所示。

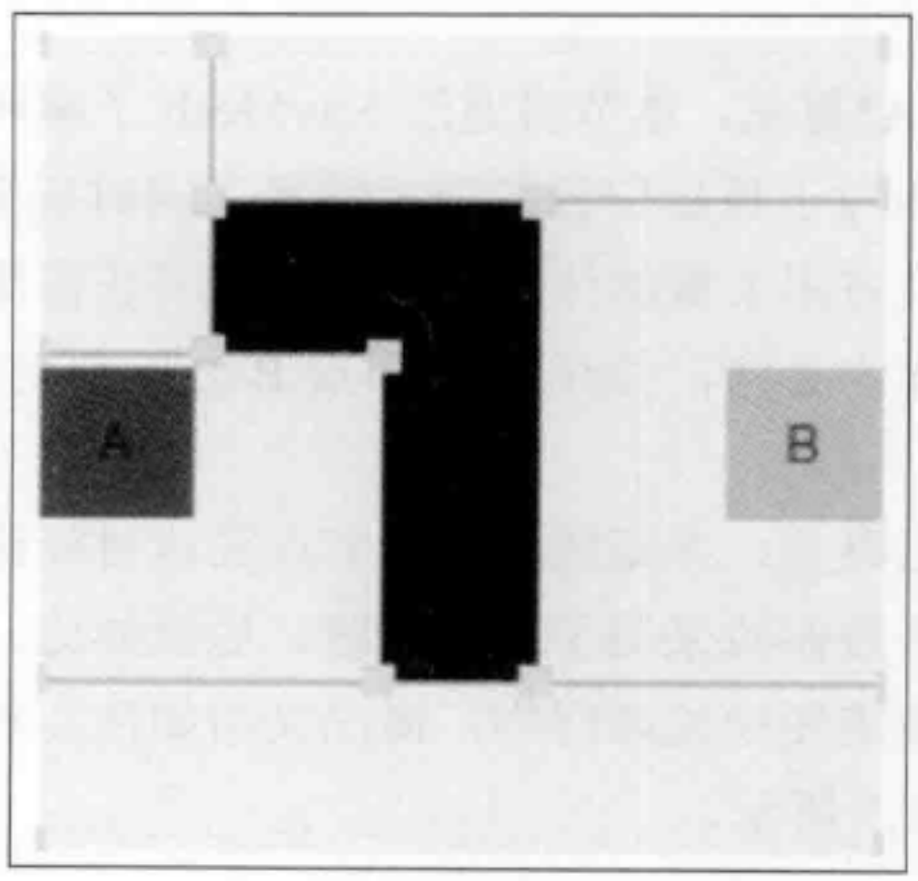

图 1.8

导航网格使用凸多边形表示 AI 实体行进的地图区域。与路点系统相比，导航网格的优点在于可生成与环境相关的信息。由于已了解了 AI 实体行进的安全区域，因而可安全地调整路径。导航网格的另一个优点是，可针对不同的 AI 实体使用同一网格。具体而言，不同的 AI 实体包含了不同的属性，例如尺寸、速度和运动能力。当一组路点经过精心设计后，对于飞行生物或 AI 控制的车辆对象，实际效果可能并不理想。对于这一类情况，导航网格可节省大量的时间。

根据某一场景采用程序方式生成导航网格其处理过程相对复杂。作为高级组件，Unity 3.5 中内置了导航网格生成器，但 Unity 5 个人版中免费提供了这一组件。第 4 章将讨论 Unity NavMesh 在游戏中的应用，并进一步阐述基于 Unity 5 的改进方案。

1.8　群 集 方 案

大量生物采用群集方式实现迁徙、捕猎或觅食等行为，例如鸟类、鱼和昆虫。与单独行动相比，这将使得群体变得更加安全、强大。对于鸟群在天空中盘旋这一类场景，设计独立飞鸟的运动和动画行为将消耗动画设计师大量的时间。如果针对飞鸟个体使用某一简单的行进规则，即可针对包含复杂、全局行为的鸟群实现整体智能效果。

类似地，对于行走或驾驶车辆的人群，同样可将集群视为独立实体进行建模，而非将个体作为主体对象进行建模。集群中的个体仅需了解当前群体的行进方向及其最近邻居的行为即可，进而发挥系统的局部机能。

1.9　行　为　树

行为树则是体现 AI 主体对象之后隐藏的控制和逻辑，并在 AAA 游戏中变得越发流行，例如《光晕》和《孢子》。前述内容简要地讨论了 FSM 机制，并提供了简单、高效的主体行为方式，即不同的状态以及状态间的转换。然而，由于 FSM 在后续操作中缺少一定的灵活性，且需要大量的人工设置，因而难以实现规模化操作。考虑到需要加入多个状态，连接大量的转换以支持全部方案，并以此考察主体对象，因此应采取一种可扩展的方法处理大型问题，这也是行为树的产生原因。

行为树表示为节点集，并通过层次结构方式予以组织。其中，节点连接至父节点，

而非彼此连接的多个状态。该结构类似于树枝状结构，行为树的名称也由此而来。

行为树的基本元素表示为任务节点，相比之下，在 FSM 中，状态则表示为主元素。相应地，存在多项不同的任务，例如 Sequence、Selector 和 Parallel Decorator。跟踪其全部功能实现较为困难，理解这一概念的最佳方式是考察相关示例。下面将转换和状态分解为多项任务，如图 1.9 所示。

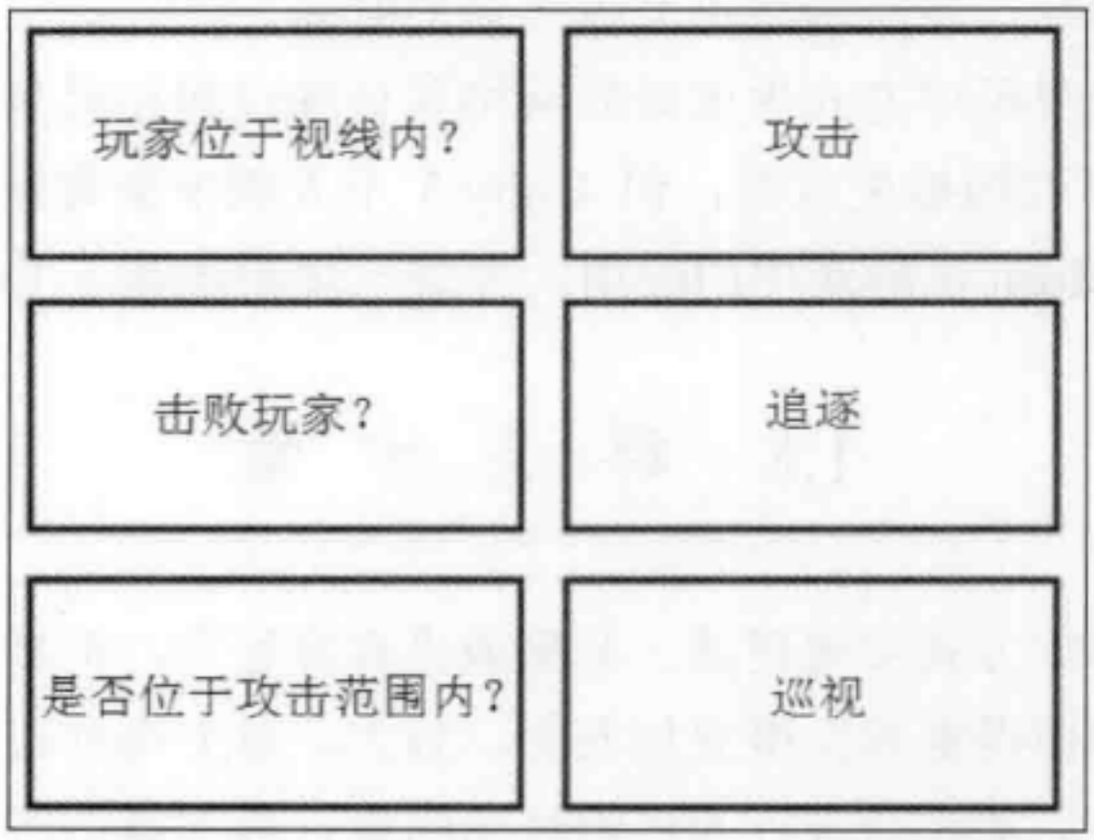

图 1.9

下面针对当前行为树考察 Selector 任务。在图 1.10 中，Selector 任务通过圆形和问号表示。其中，Selector 按照自左至右的顺序计算各个子节点。首先，Selector 选择攻击玩家，如果 Attack 任务返回成功标识，则 Selector 任务执行完毕并返回至父节点处（如果存在父节点）。如果 Attack 任务无效，则 Selector 将尝试 Chase 任务。如果 Chase 任务无效，则会继续尝试 Patrol 任务。图 1.10 显示了该树形概念的基本结构。

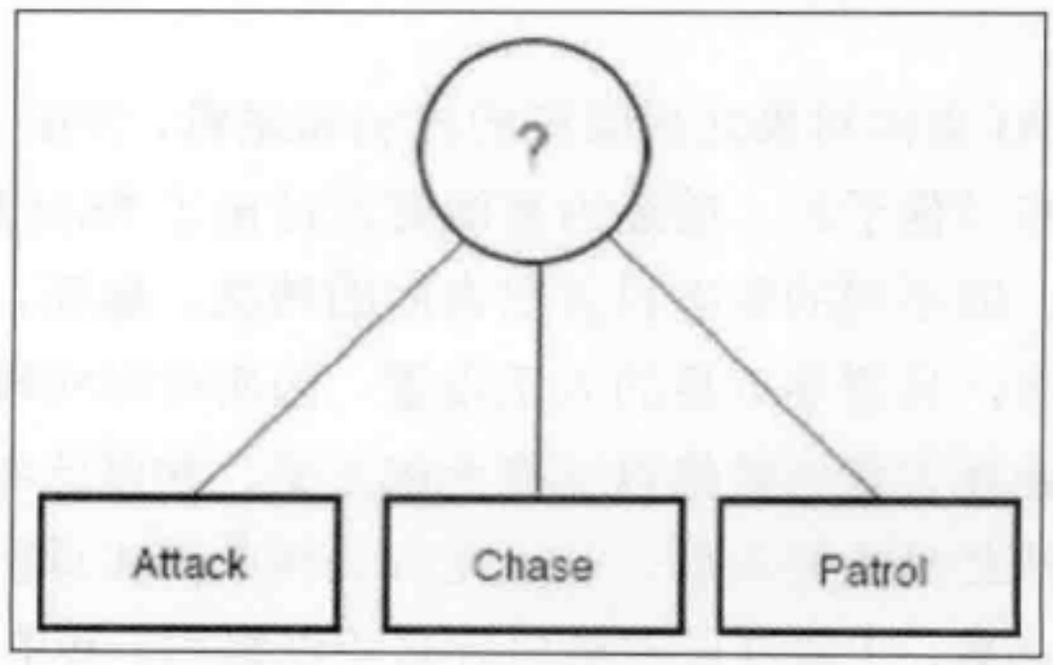

图 1.10

测试也是行为树中的任务之一。图 1.11 显示了 Sequence 任务的具体应用，相关任务采用包含箭头的矩形表示。另外，根 Selector 选择了第一个 Sequence 动作(action)。Sequence 动作的首项任务即是检测玩家角色是否位于攻击范围内。若该项任务成功执行，将会继续处理下一项任务，即攻击玩家角色。若 Attack 任务也成功返回，则全部序列将返回成功状态，Selector 将以当前行为告终，且不会继续执行其他 Sequence 任务。若邻近检测任务无效，Sequence 动作将不会执行 Attack 任务，并向父 Selector 任务返回无效状态。随后，Selector 将选择当前序列中的下一项任务，即“Lost or Killed Player?”。

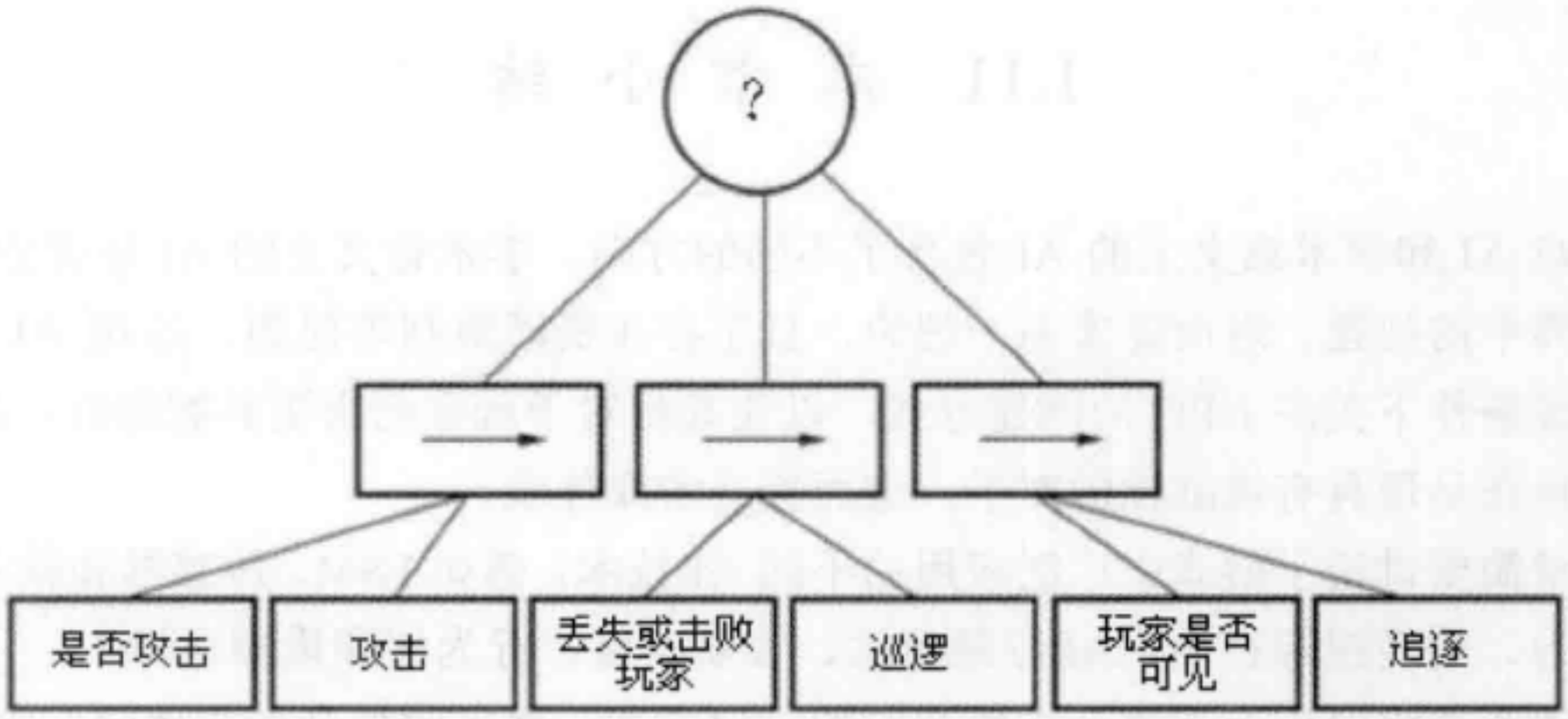

图 1.11

Parallel 任务和 Decorator 则是其他两个较为常见的组件。其中，Parallel 任务同时处理全部子节点，而 Sequence 和 Selector 任务仅是逐一处理其子节点任务。Decorator 则是另外一种任务类型，并可调整自身子节点任务的行为，包括是否运行其子节点任务以及运行次数等问题。第 6 章将讨论 Unity 中基本行为树实体的实现方式。

1.10　模 糊 逻 辑

简而言之，模糊逻辑体现了一类近似结果，而非二元结论。相应地，可采用模糊逻辑和推理以使 AI 行为更具真实感。

下面采用第一人称射击类游戏中的敌方士兵角色作为主体对象，并以此描述这一

基本概念。无论采用有限状态机或行为树，当前主体对象均需要判断“应移至 x、y 或 z 状态？”“当前任务返回 true 或 false？”这一类问题。在缺少模糊逻辑的支持下，仅可通过二元值判断上述问题的答案。例如，当判断敌方角色是否看见玩家时，仅存在是/否两种结论。然而，如果对当前决策实施进一步处理，则可获得更为有趣的行为方式。例如，当士兵发现玩家后，可询问“自己”是否配备了足够多的弹药射杀玩家；或者中弹后是否存在足够的血值而得以存活；或者在攻击玩家过程中判断玩家周围是否存在其他盟友。综上所述，AI 角色的行为将变得越发有趣，战局将变得更加变幻莫测，游戏体验也将呈现更高的可信度。

1.11 本 章 小 结

游戏 AI 和学术意义上的 AI 包含了不同的方向。学术意义上的 AI 研究尝试解决现实世界中的问题，进而证实某一理论，且不存在资源限制等问题。游戏 AI 则在有限的资源条件下关注 NPC 的创建方式，以使其相对于玩家而言更具智能性。游戏 AI 的目标旨在呈现具有挑战性的对手，进而提升游戏体验。

本章简要讨论了游戏中广泛应用的不同 AI 技术，例如 FSM、传感器和输入系统、群集行为、路径跟踪行为、AI 寻路算法、导航网格、行为树和模糊逻辑。

后续章节将讨论上述概念在游戏中的应用方式，并以此提升游戏体验。第 2 章将实现 FSM，其中涉及主体对象、状态等概念，及其在游戏中的应用方式。

第 2 章　有限状态机

本章将对 FSM 模式及其在游戏中的应用进行扩展，并讨论如何在简单的 Unity 游戏中实现有限状态机。对此，本章将创建一个敌方坦克角色，并讲解其示例代码和项目中的各个组件。该项目包含下列内容：

- 理解 Unity 中的状态机的调整。
- 创建状态和转换。
- 通过示例构建场景。

Unity 5 引入了有限状态机行为，并可视为 Mecanim 动画（Unity 4.x 版本后被引入）状态的一般扩展。然而，新增的状态机行为与动画系统无关，本章将利用这一类新特性快速实现基于状态的 AI 系统。

在游戏中，玩家可控制坦克对象，而敌方坦克角色通过 4 个路点在场景中运动。一旦玩家坦克进入视见范围后，敌方角色即开始进行追逐。当处于攻击范围内，即开始对坦克主体对象射击。这一简单示例通过有趣的方式让读者了解 AI 以及 FSM。

2.1　FSM 应用

尽管本章主要讨论 FSM 并以此实现游戏中的 AI 系统，但需要指出的是，FSM 广泛地应用于游戏以及软件设计和程序设计中。实际上，Unity 5 中的新系统首先应用于 Mecanim 动画系统中。

在日常生活中，我们可将众多事物归类于多种状态。在程序设计中，高效的模式即是对真实设计中的简单性予以模拟，这一点也体现于 FSM 中。通过观察可知，诸多事物均可处于某一状态下，例如，周边环境是否安装了灯泡？此处，灯泡可能处于两种状态，即开启或关闭状态。在中学时曾经学习到，物体也可处于不同的状态，例如水可以呈现为固态、液态和气态。类似于程序设计中的 FSM，变量可触发状态变化，对于当前示例，热量促使水分在不同状态间转化，如图 2.1 所示。

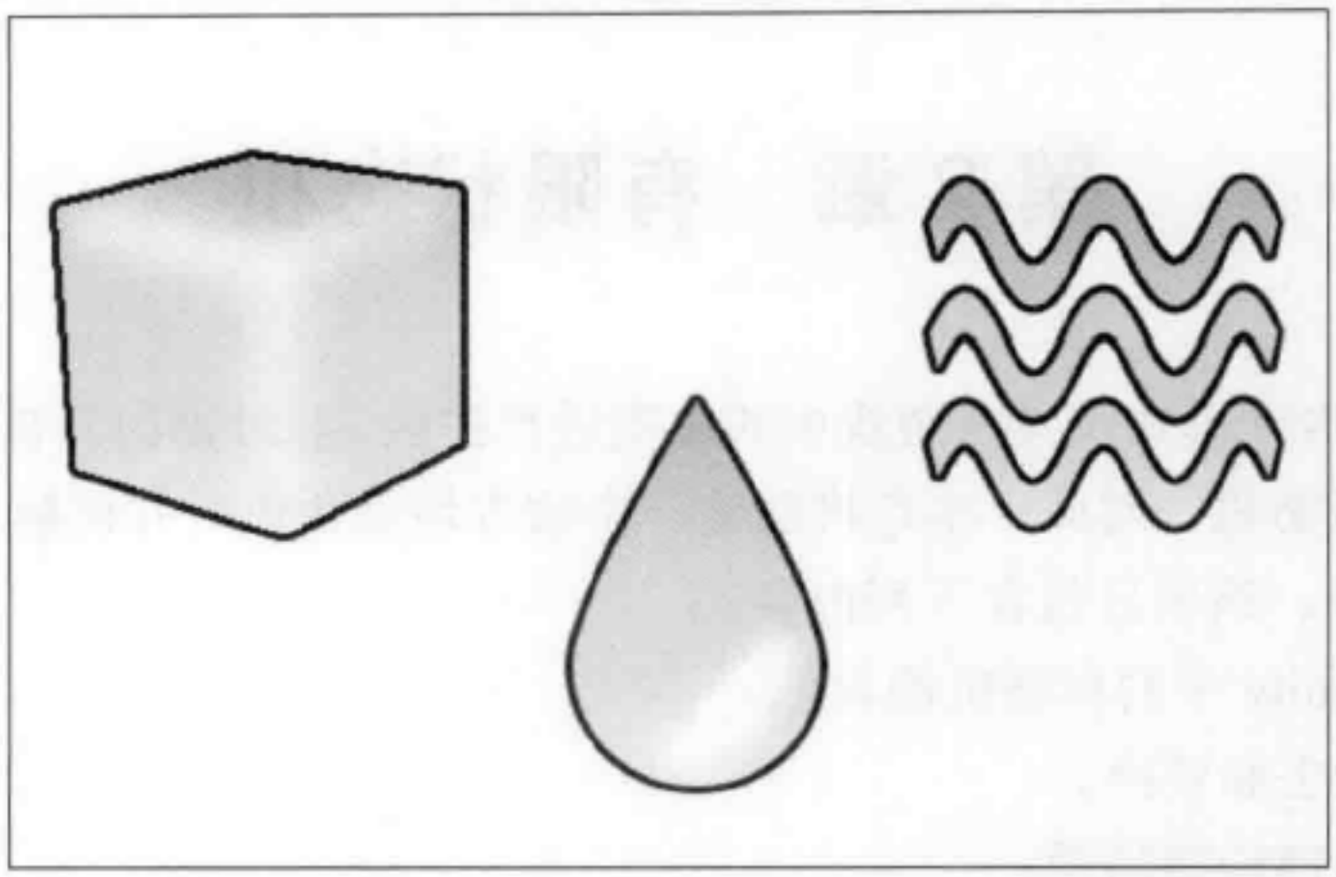

图 2.1

虽然程序设计模式实现并无硬性规定，但某一时刻仅包含单一状态则是 FSM 的一个重要特征，这表示，两个状态间的转换方式包含了多种切换，并可实现“冰逐渐融化成水”这一类效果。另外，主体对象可包含多个 FSM，驱动任意数量的行为，且状态中甚至可包含自身的状态机。利用状态机，读者甚至可实现电影《盗梦空间》中的梦幻场景。

2.2 生成状态机行为

相信读者已对状态机这一概念有所了解，下面讨论其具体实现过程。

在 Unity 5.0.0f4 版本中，状态机仍隶属于动画系统。尽管如此，状态机依然具有足够的灵活性，且实现过程中并不涉及动画内容。如果读者看到相关代码引用了 Animator 组件或 AnimationController 数据包，将其视为当前实现的某种技巧即可。相信 Unity 在其后续版本中将解决这一问题，但对应的概念未发生任何变化。

下面将在 Unity 中尝试创建新的项目。

2.2.1 生成 AnimationController 资源

AnimationController 资源包表示为 Unity 中的一种资源类型，用于处理状态和转换。实际上，该资源包可视为 FSM，但其功能远不止于此，本节主要讨论其功能项中

的 FSM 部分。这里，动画控制器可通过 Assets 菜单予以生成，如图 2.2 所示。

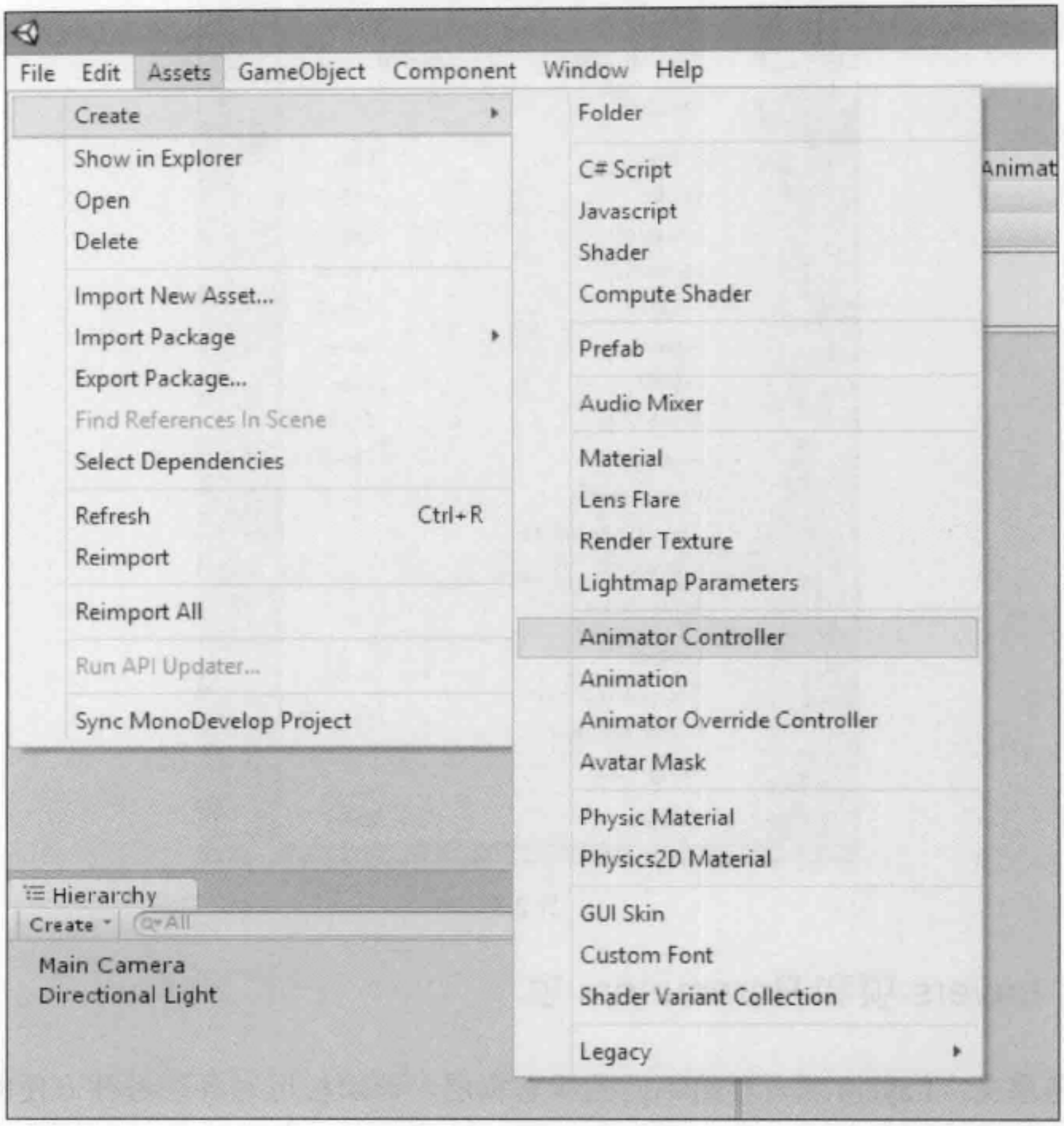

图 2.2

当动画控制器创建完毕后，将出现于项目的资源文件夹中，此处将其命名为 TankFsm。与其他资源类型不同，当选择了动画控制器后，层次结构将呈现为空状态。其原因在于，动画控制器使用自身的窗口。用户可在层次结构中简单地单击 Open 并打开 Animator 窗口，或者在 Windows 菜单中打开窗口，如图 2.3 所示。

提示：此处应确保选择Animator而非Animation，二者具有完全不同的窗口和特征。

在继续讲解之前，下面首先对上述各窗口加以讨论。

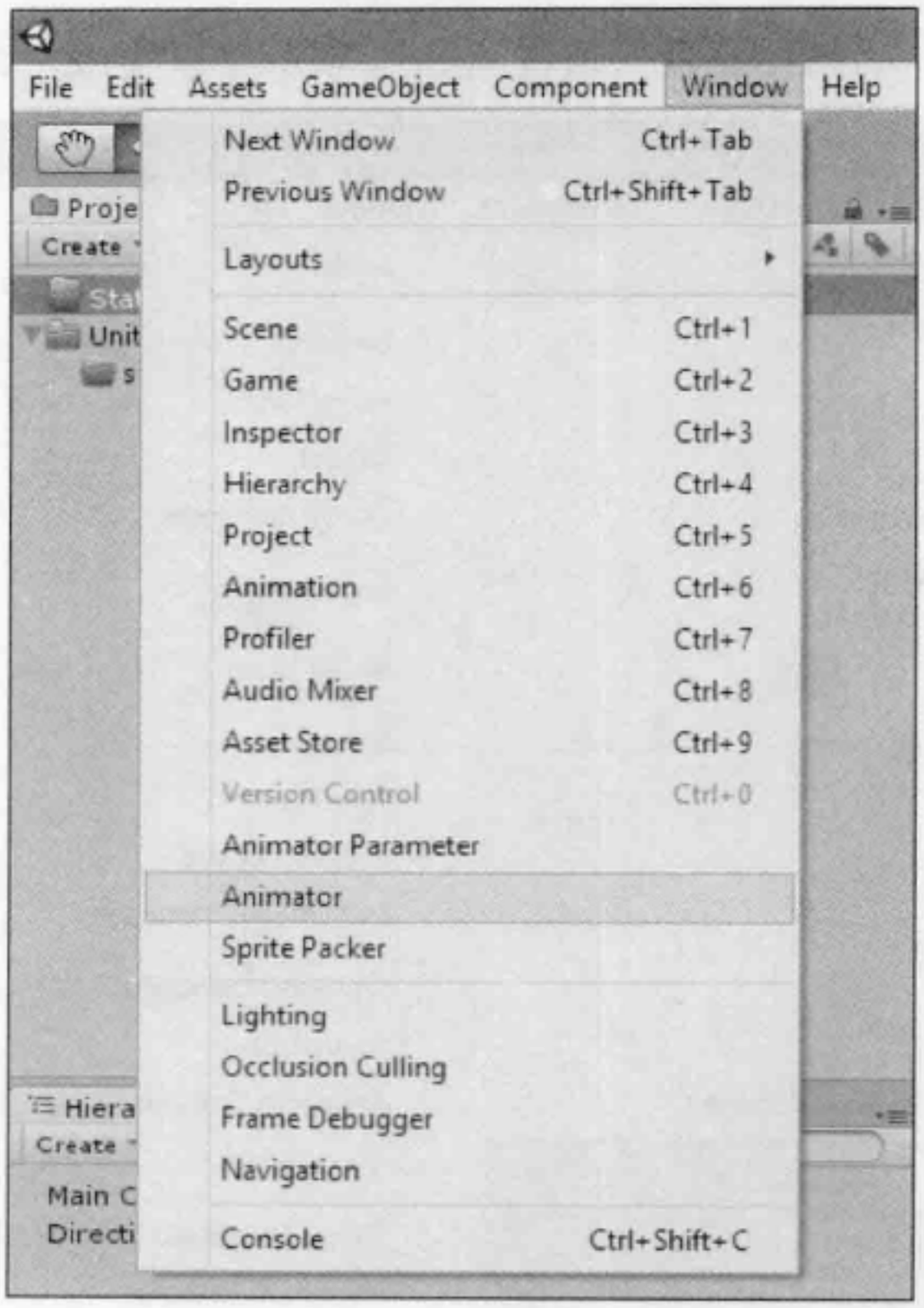

图 2.3

2.2.2　Layers 项和 Parameters 项

顾名思义，Layers 项可存储不同的状态机层，该面板可对各层进行方便的组织，并包含了相应的视觉表达效果。考虑到主要与动画有关，因而此处并不打算对该面板做过多讨论，但读者依然需要对该面板有所了解。关于 Layers 层的对应窗口，如图 2.4 所示。

相关项的内容如下。

- Add layer：该按钮在列表下方创建新层。
- Layer list：表示为动画控制器内的当前层。用户可点击并选取某一层，并拖曳相关层以对其进行重新排列。
- Layer settings：针对当前层包含了与动画相关的设置。

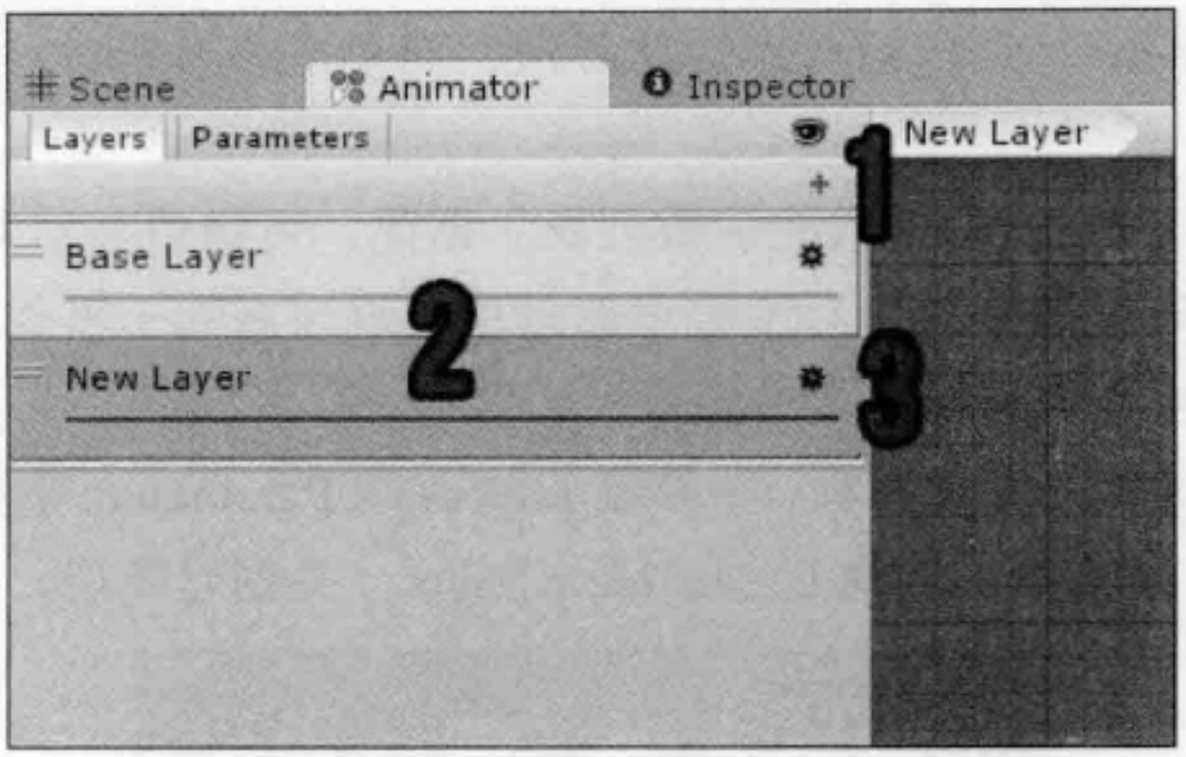

图 2.4

另外，Parameters 面板与动画控制器的应用紧密相关。此处，参数表示为确定状态间转换时的变量，并可通过脚本对其加以访问，进而对状态加以驱动。这里存在 4 种参数：float 型、int 型、bool 型以及 trigger 型。作为 C#语言中的基础类型，相信读者已对前 3 种类型有所了解，而 trigger 则与动画控制器相关，读者不应将其与物理触发器混淆。此处，trigger 表示为显式触发状态间转换的方式。

图 2.5 显示了 Parameters 面板中的各个元素。

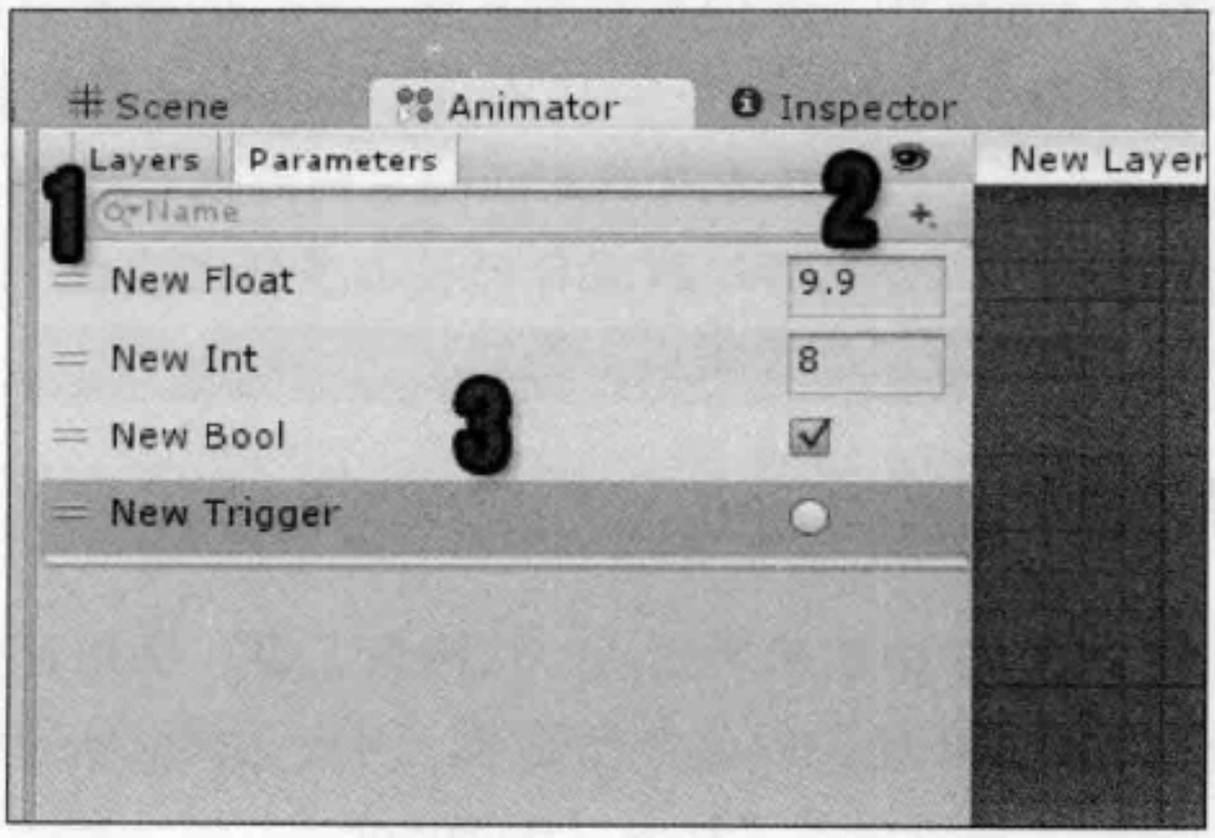

图 2.5

其中相关内容包括以下方面。

- Search：可用于快速搜索参数。对此，可简单地输入名称，随后将显示搜索结果列表。

- Add parameter：该按钮可添加新参数。当单击该按钮时，须选择相应的参数类型。
- Parameter list：表示生成的参数列表。据此，可赋值并对其进行查看。另外，用户还可通过拖曳操作对参数重新排序。这可视为一种组织方式，且不会对功能性产生任何影响。

最后，用户还可单击眼球图标，并隐藏 Layers 和 Parameters 面板。当关闭面板时，仍可单击 Layers 下拉菜单，选择 Create New Layer，并创建新层，如图 2.6 所示。

图 2.6

2.2.3　动画控制查看器

动画控制查看器与 Unity 中一般的查看器不同。常规的查看器可向游戏对象添加组件，而动画控制查看器中包含了一个名为 Add Behaviour 的按钮，并可向其加入 StateMachineBehaviour，这也是两种查看器之间的主要差别。除此之外，动画控制查看器还可针对所选状态、子状态、转换或混合树显示序列化信息，这一点与基于所选游戏对象及其组件的常规查看器的数据显示类似。

2.2.4　行为的图像化

状态机行为则是 Unity 5 中的新增概念。从概念上讲，尽管状态存在于 Mecanim 的原始实现中，但转换则是在场景后台进行处理，用户无须对状态的进入、转换和退出进行过多干涉。Unity 5 通过引入“行为（Behavior）”这一概念对此提供了解决方案，并通过内建功能处理常见的 FSM 逻辑。

通过表面的字义，读者可能认为“行为”与 MonoBehaviour 有关，但二者仅存在少许关联。实际上，行为源自 ScriptableObject，而非 MonoBehaviour，且仅作为资源包存在，因而无法置于场景中，或作为组件添加至 GameObject 中。

2.2.5　生成第一个状态

Unity 在动画控制器中生成了多个默认状态，包括 New State、Any State、Entry 和 Exit。相关操作包括：

- 用户可通过单击对应状态，进而在该窗口内实现状态的选取；同时，还可将其拖曳至画布中的任意位置，并对状态进行移动。
- 选择名为 New State 的状态，通过右击鼠标并单击 Delete 项，或者按 Delete 键删除该状态。
- 当选取了 Any State 状态后，用户将会发现无法将其删除，对于 Entry 状态也是如此。这一类状态在动画控制器中不可或缺，并包含独特的应用方法，如图 2.7 所示。

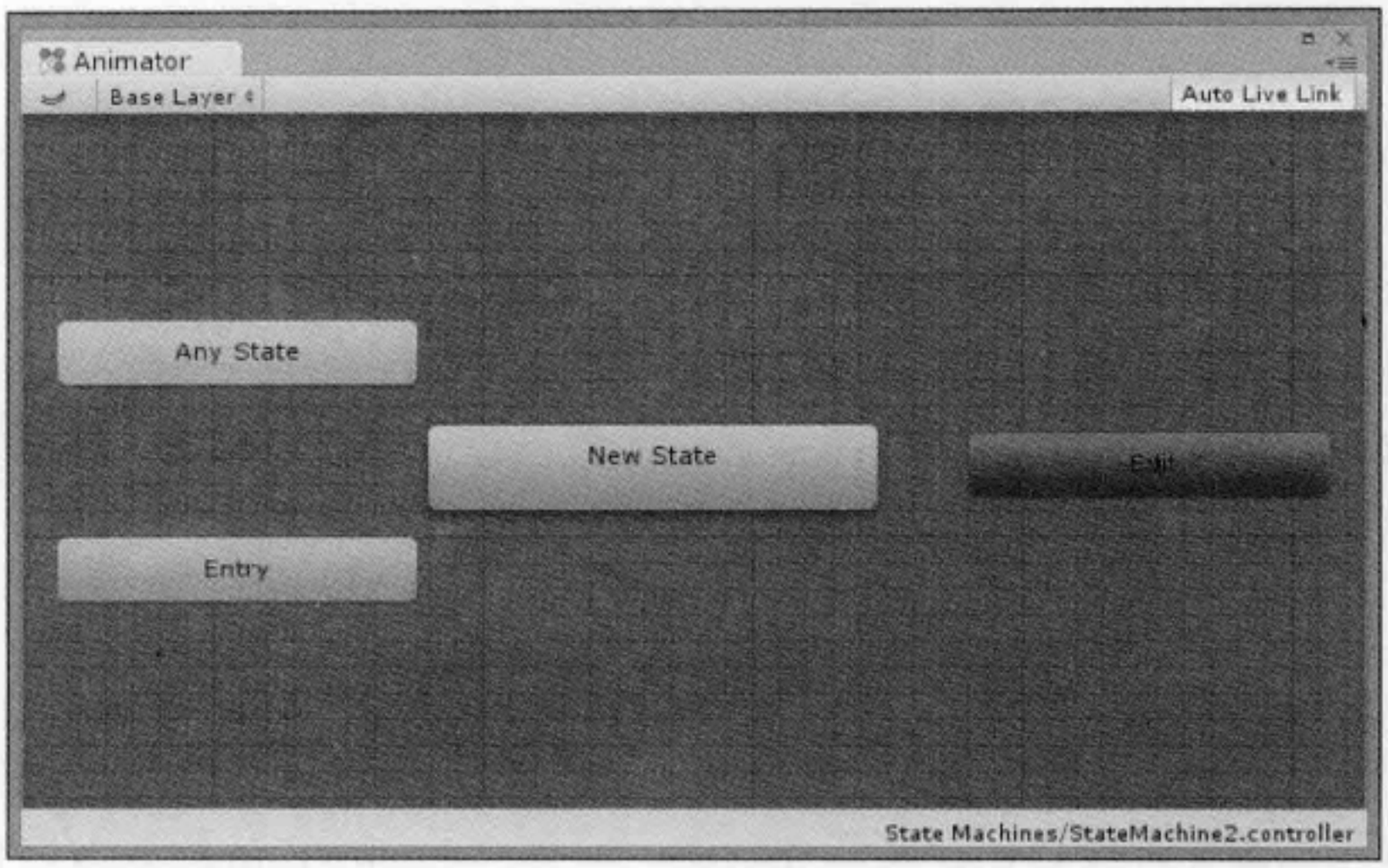

图 2.7

当生成首个状态时，可点击画布并于随后选择 Create State，这可打开多个选项。当前，可从中选择 Empty 选项。其他两项内容（From Selected Clip 和 From New Blend Tree）暂时不会用于当前项目中。

2.2.6　状态间的转换

当生成自定义状态时，可看到一个箭头连接至 Entry 状态，并呈现为橘黄色。Unity

将默认状态设置为橘黄色，以区别于其他状态。当仅存在一个状态时，该状态将被自动选取为默认状态，并自动连接至 Entry 状态。相应地，通过右击该状态，并于随后单击 Set as Layer Default State，还可实现人工方式的默认状态的选取。随后，对应状态将变为橘黄色，Entry 状态将自动与其进行连接。其中，连接箭头表示为转换连接器，并对转换出现的方式和时机加以控制。然而，Entry 状态至默认状态间的连接器则较为独特，由于对应转换自动生成，因而未提供任何选项。

通过右击状态节点，并于随后选择 Make Transition，即可通过人工方式在状态间赋予转换。这将生成所选状态和鼠标箭头之间的转换。当选择了转换目标后，可简单地单击目标节点即可。需要注意的是，用户无法对转换进行重定向操作。对此，也希望 Unity 后续版本支持此项功能。当前用户可通过如下方式移除某一转换：选择该转换并予以删除，并于随后通过人工方式赋以新的转换。

2.3 创建玩家坦克对象

打开本书配套资源中的示例项目。

在项目文件夹中，可对资源包进行分组，以使其安排有序。例如，可将状态机置于名为 StateMachines 的文件夹中。本章提供的资源包已经分组完毕，因而读者可将后续操作创建的资源包和脚本拖曳至对应文件夹内。

2.4 生成敌方坦克对象

下面将在资源文件夹内创建动画控制器，即称作 EnemyFsm 的敌方坦克状态机。

该状态机将驱动坦克的基本动作，如前所述，在当前示例中，敌方角色可巡行、追逐并向玩家射击。当设置状态机时，可选择 EnemyFsm 资源包并打开 Animator 窗口。

下面将创建 3 个空状态，在概念上和功能上表示敌方坦克状态，并分别将其命名为 Patrol、Chase 和 Shoot。待创建和命名完毕后，应确保赋予了正确的默认状态。当前，这将根据状态的创建和命名顺序而变化，当前需要将 Patrol 状态设置为默认状态，因而可单击该状态并选取 Set as Layer Default State。随后，该状态变为橘黄色，并与 Entry 状态连接。

2.4.1　选择转换

针对状态间的走向，此处需要制定相应的设计和逻辑决策。当对应转换制定完毕后，还需考虑触发此类转换的对应条件，以确保符合逻辑并根据设计观点进行工作。当读者在自己的项目中使用这一类技术时，不同因素将对上述转换的处理方式产生显著的影响。为了较好地描述这一话题，可将对应转换设计得简单而富有逻辑性，其中包括以下方面。

- Patrol 状态：从该状态可转换至追逐状态。对此，可使用条件链选择即将转换的状态（若存在的话）。例如，敌方坦克是否可看到玩家？若是，则进入下一个步骤；否则，敌方坦克将继续处于巡视状态。
- Chase 状态：根据该状态，可持续检测玩家是否位于视线内，并保持巡视状态。如果处于攻击范围内，则敌方坦克开始射击；若玩家消失于视线之外，则敌方坦克将再次返回巡视状态。
- Shoot 状态：同样，可对射击和视线范围进行检测，进而确定是否进行追逐。

上述示例包含了简单而清晰的转换规则集。如果连接对应状态，其状态图如图 2.8 所示。

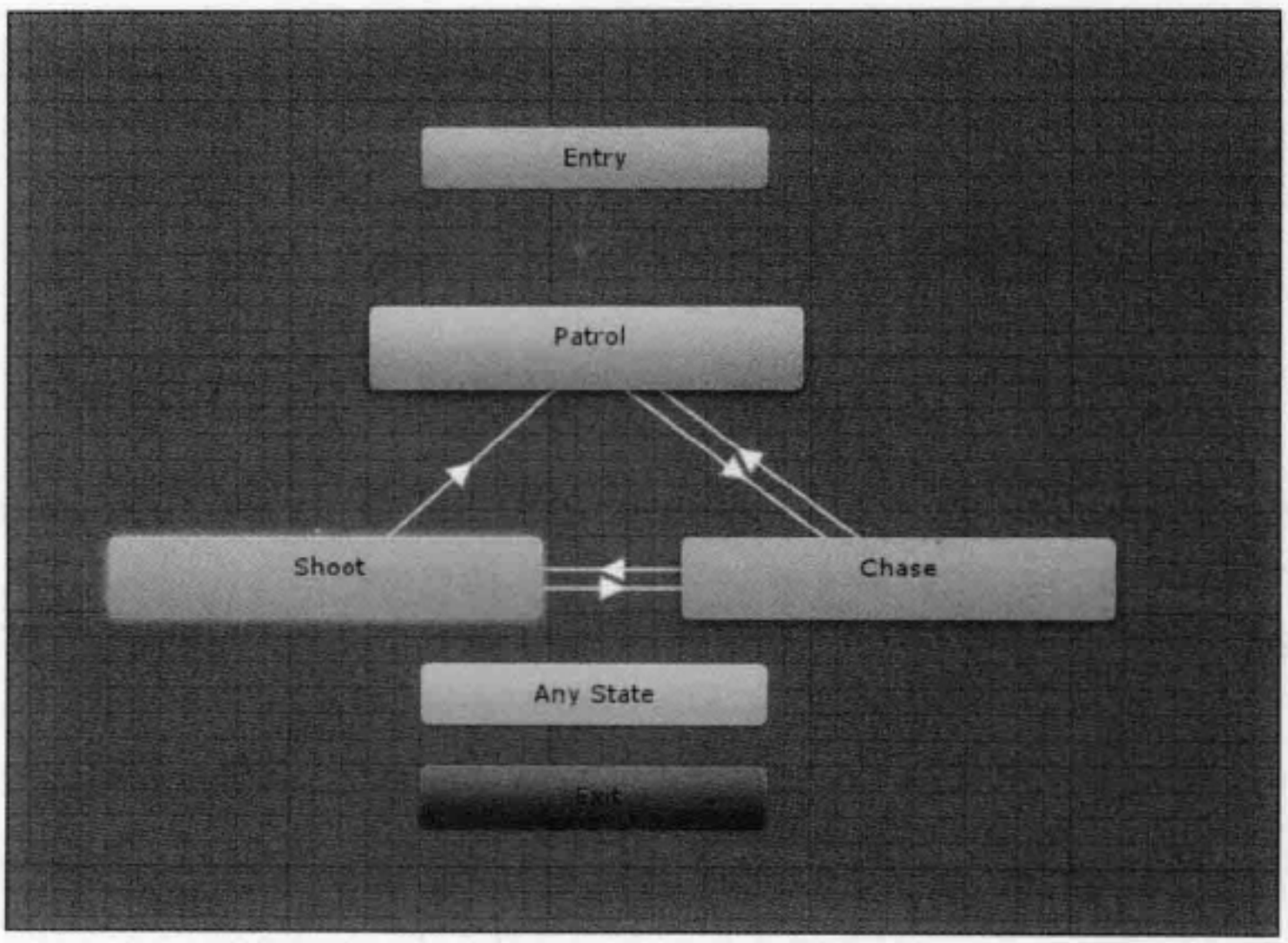

图 2.8

需要说明的是，节点的替换操作取决于用户，且不会对状态机的功能产生任何影响。读者可尝试按照有序方式设置节点，并从视觉角度上对转换进行跟踪。

至此，对应状态已制定完毕，下面将对其赋予相关行为。

2.4.2　实现过程

目前，相信读者已经了解了 FSM 中状态间的逻辑连接，本节讨论其编码实现。首先可选取 Patrol 状态。在层次结构中，读者可看到标记为 Add Behaviour 的按钮，单击该按钮将出现快捷菜单，这一点与常规游戏对象的 Component 按钮十分相似。如前所述，该按钮将生成独有的状态机行为。

随后可将该行为命名为 TankPatrolState，这将在文件夹内生成同名的脚本，并将其绑定至所创建的状态中。通过项目窗口，或者双击查看器中的脚本名称即可对该脚本进行查看。对应内容如下所示：

```
using UnityEngine;
using System.Collections;

public class TankPatrolState : StateMachineBehaviour {

    //OnStateEnter is called when a transition starts and the state
machine starts to evaluate this state
    //override public void OnStateEnter(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex) {
    //
    //}

    //OnStateUpdate is called on each Update frame between
    //OnStateEnter and OnStateExit callbacks
    //override public void OnStateUpdate(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex) {
    //
    //}

    //OnStateExit is called when a transition ends and the state
machine finishes evaluating this state
```

```
    //override public void OnStateExit(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex) {
    //
    //}

    //OnStateMove is called right after Animator.OnAnimatorMove().
Code that processes and affects root motion should be implemented here
    //override public void OnStateMove(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex) {
    //
    //}

    //OnStateIK is called right after Animator.OnAnimatorIK(). Code
that sets up animation IK (inverse kinematics) should be implemented
here.
    //override public void OnStateIK(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex) {
    //
    //}
}
```

提示：读者可访问http://www.packtpub.com下本书的示例代码。另外，读者还可访问http://www.packtpub.com/support，经注册后可通过电子邮件方式获取相关文件。

Unity 为用户生成上述文件，且全部方法均被注释掉。实际上，注释代码饰演了操作指南这一类角色。类似于 MonoBehaviour 中定义的多个方法，当前方法通过底层逻辑进行调用，读者不必了解其背后的含义，只需知晓何时对其加以调用即可。注释代码提供了各方法调用时机的简要说明，方法名也包含了自解释内容。其中，读者无须关注 OnStateIK 和 OnStateMove 方法，二者表示为动画消息，因而可对其予以删除并保存文件。

在代码注释中，相关内容包括：

- 当转换开始并进入某一状态时，OnStateEnter 将被调用。
- 在 MonoBehaviors 更新后，OnStateUpdate 在各帧内被调用。
- 状态转换结束时，OnStateExit 将被调用。

如前所述，下列两个状态与动画相关，且当前未投入使用：

- 在 IK 系统更新之前调用 OnStateIK，该方法与动画以及骨骼等概念有关。
- OnStateMove 用于 Avatar 结构，该结构用于根运动中。

另一个需要注意的重要信息则是传递至上述方法中的参数，如下所示：

- 动画参数表示为动画器的引用，且包含动画控制器以及状态机。引申开来，用户可获取动画控制器上游戏对象的引用，进而可获得与其绑定的其他组件。回忆一下，状态机行为仅作为资源包存在，且不会存在于类中。因此，最佳方式即是获取运行期类的引用，例如单一行为。
- 动画状态信息提供了与当前状态相关的信息，该应用主要集中于动画状态信息，在当前应用程序中并无太大作用。
- 最后，层索引表示为一个整数，表示当前状态位于状态机中的哪一层。其中，基准层表示为 0，其上方各层数字依次增加。

至此，相信读者已经理解了状态机行为的基本概念，下面依次查看其余组件。在实际考察相关行为之前，还需返回至状态机，并添加某些参数以驱动相关状态。

1. 设置条件

敌方坦克对象包含了某些条件，并以此实现状态转换。此类实际参数负责对各项功能进行驱动。

下面首先考察 Patrol 状态。为了使敌方坦克对象从 Patrol 状态转换至 Shoot 状态，需要检测敌方角色与玩家之间的距离，该值可通过浮点值进行描述。因此，在 Parameters 面板中，可添加一个浮点值，并将其命名为 distanceFromPlayer。除此之外，还可利用该参数判断是否进入 Chase 状态。

Shoot 状态和 Chase 状态共享公有条件，即玩家是否可见。对此，可通过简单的光线投射进行判断，进而确定玩家是否位于视线内。对于 Boolean 参数，经创建后可将其命名为 isPlayerVisible，此处可将其勾选掉，以使其处于 False 状态。

当前，可通过转换连接器的查看器对条件进行赋值。对此，可简单地选取某一连接器，查看器将显示与当前转换相关的信息，以及某些更加重要的条件信息，并通过列表加以显示。当添加条件时，可简单地单击+图标，如图 2.9 所示。

下面将逐一处理各个转换。

- Patrol 状态转换为 Chase 状态：
  - distanceFromPlayer < 5。
  - isPlayerVisible == true。

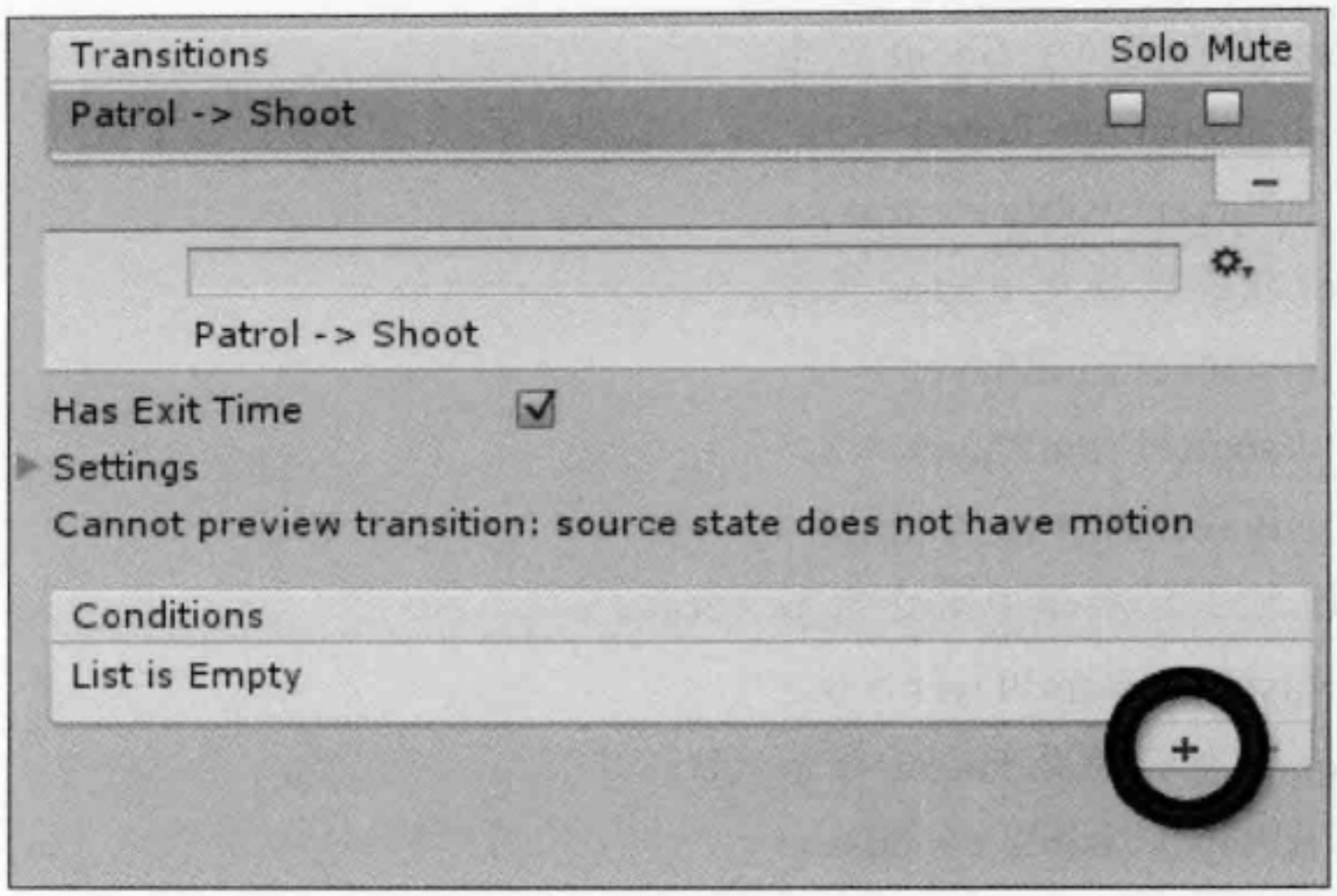

图 2.9

最终设置结果如图 2.10 所示。

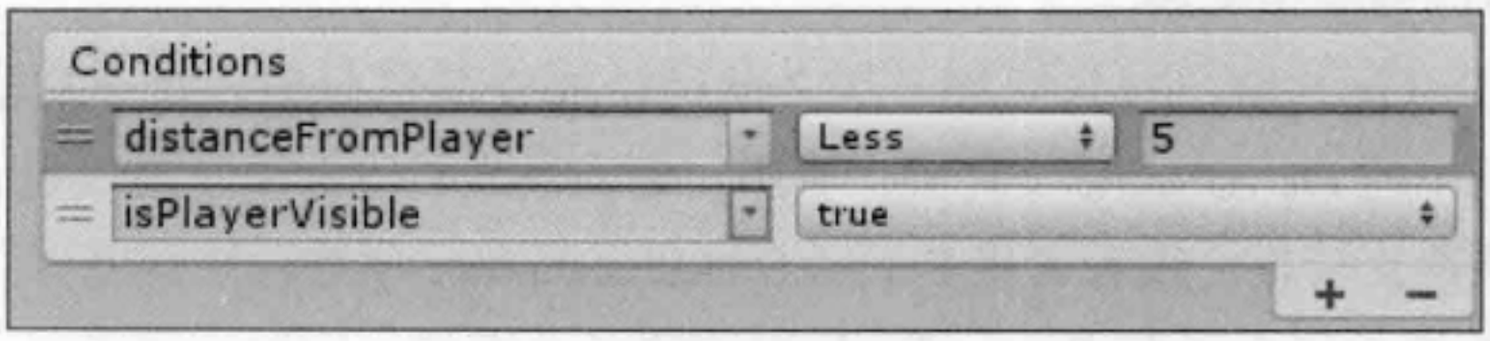

图 2.10

由于包含了两个独立的条件并可触发某一转换，因而 Chase 状态至 Patrol 状态的转换则较为有趣。如果简单地向该转换添加两个条件，则二者均针对所产生的转换设置为 true。但此处需要检测玩家是否位于范围之外，或视线之外。幸运的是，两个状态间可包含多个转换，因而可像往常一样添加另一个转换连接。相应地，可右击 Chase 状态，并生成一个指向 Patrol 状态的转换。不难发现，查看器上方列出了两个转换。除此之外，转换连接指示符显示了多个箭头（而非单一箭头），进而表明两个状态间存在多个转换。在查看器中选取各转换将生成各独立条件，如下所示：

- Chase 状态转换至 Patrol 状态（A）。
  - distanceFromPlayer > 5。
- Chase 状态转换至 Patrol 状态（B）。
  - isPlayerVisible == false。

- Chase 状态转换至 Shoot 状态。
  - distanceFromPlayer < 3。
  - isPlayerVisible == true。
- Shoot 状态转换至 Chase 状态。
  - distanceFromPlayer > 3。
  - distanceFromPlayer < 5。
  - isPlayerVisible == true。
- Shoot 状态转换至 Patrol 状态（A）。
  - distanceFromPlayer > 6。
- Shoot 状态转换至 Patrol 状态（B）。
  - isPlayerVisible == false。

当前，状态和转换已经设置完毕，接下来需要编写驱动上述各值的脚本。此处，全部工作即是设置上述各值，状态机将会处理其他值。

2. 通过代码驱动参数

在进一步讨论之前，首先回顾一下之前讨论的资源包问题。对于初学者，可打开本章配套资源中的 DemoScene 文件夹。其中，对应场景较为简单，且仅包含了场景预置组件以及某些路点转换，读者可将 EnemyTankPlaceholder 预置组件拖曳至当前场景中。

此时，场景中包含了与 EnemyTank 相关的多个组件，同时，读者也可再次对第 4 章中的 NavMesh 和 NavMeshAgent 进行回顾。为了保证项目的正常运行，当前各组件均不可或缺。此处主要关注 Animator 组件，并装载之前创建的状态机（即动画控制器）。对此，在执行后续操作之前，可将状态机拖曳至空选项框中。

除此之外，还需针对玩家对象采用占位符，对此可拖曳 PlayerTankPlaceholder 预置组件。类似于敌方坦克占位符预置组件，玩家坦克占位符预置组件也包含了若干组件，当前操作可暂时对其予以忽略。对此，可简单地将其拖曳至场景中，并执行后续操作。

随后，需要向 EnemyTankPlaceholder 游戏对象添加新的组件，即 TankAi.cs 脚本，该脚本文件位于本书配套资源的 Chapter 2 文件夹内。打开该脚本文件后，对应内容如下所示：

```
using UnityEngine;
using System.Collections;
```

```
public class TankAi : MonoBehaviour {
    //General state machine variables
    private GameObject player;
    private Animator animator;
    private Ray ray;
    private RaycastHit hit;
    private float maxDistanceToCheck = 6.0f;
    private float currentDistance;
    private Vector3 checkDirection;

    //Patrol state variables
    public Transform pointA;
    public Transform pointB;
    public NavMeshAgent navMeshAgent;

    private int currentTarget;
    private float distanceFromTarget;
    private Transform[] waypoints = null;

    private void Awake() {
        player = GameObject.FindWithTag("Player");
        animator = gameObject.GetComponent<Animator>();
        pointA = GameObject.Find("p1").transform;
        pointB = GameObject.Find("p2").transform;
        navMeshAgent = gameObject.GetComponent<NavMeshAgent>();
        waypoints = new Transform[2] {
            pointA,
            pointB
        };
        currentTarget = 0;
        navMeshAgent.SetDestination(waypoints[currentTarget].
position);
    }

    private void FixedUpdate() {
        //First we check distance from the player
```

```
        currentDistance = Vector3.Distance(player.transform.position,
transform.position);
        animator.SetFloat("distanceFromPlayer", currentDistance);

        //Then we check for visibility
        checkDirection = player.transform.position - transform.
position;
        ray = new Ray(transform.position, checkDirection);
        if (Physics.Raycast(ray, out hit, maxDistanceToCheck)) {
            if(hit.collider.gameObject == player){
                animator.SetBool("isPlayerVisible", true);
            } else {
                animator.SetBool("isPlayerVisible", false);
            }
        } else {
            animator.SetBool("isPlayerVisible", false);
        }

        //Lastly, we get the distance to the next waypoint target
        distanceFromTarget = Vector3.Distance(waypoints[currentTarget].
position, transform.position);
        animator.SetFloat("distanceFromWaypoint", distanceFromTarget);
    }

    public void SetNextPoint() {

        switch (currentTarget) {
            case 0:
                currentTarget = 1;
                break;
            case 1:
                currentTarget = 0;
                break;
        }
        navMeshAgent.SetDestination(waypoints[currentTarget].
```

```
position);
    }
}
```

运行上述脚本的相关变量已定义完毕，下面依次对其功能加以考察，如下所示。

- GameObject player：表示玩家占位符预置组件的引用。
- Animator animator：表示敌方坦克动画器，其中包含了所创建的状态机。
- Ray ray：在 FixedUpdate 循环中，用于光线投射测试的光线声明。
- RaycastHit hit：在光线投射测试中得到的碰撞信息声明。
- Float maxDistanceToCheck：该值与状态机转换中设置的对应值保持一致。实际上，当前仅检测玩家距离。除此之外，假设玩家对象位于有效范围之外。
- Float currentDistance：表示为玩家对象和敌方坦克之间的当前距离。

上述内容忽略了某些变量，稍后将对其加以讨论。这一类变量一般用于巡视状态中。

Awake 方法可获得指向玩家和动画器变量的引用。另外，上述各变量还可声明为 public 类型，或者采用[SerializeField]前缀属性，并通过查看器进行设置。

FixedUpdate 方法较为直观，代码的第一部分内容获取玩家对象和敌方坦克间的距离。其中应注意 animator.SetFloat("distanceFromPlayer",currentDistance)部分，该方法调用将脚本中的信息传递至状态机中定义的参数中。代码的后续部分也基本类似，也就是说，传递 Boolean 类型的光线投射碰撞结果。最后，代码设置 distanceFromTarget 变量，以供后续巡行状态使用。

不难发现，代码自身并未涉及与转换相关的状态机处理方式，且仅仅传递状态机所需的信息，并由状态机负责处理其余内容。

3. 移动敌方坦克对象

通过观察可知，除了上述变量之外，代码中并未包含坦克的运动逻辑。该过程可通过子状态机（即某一状态中的状态机）轻松地加以处理。初看之下，这一概念令人难以理解，此处可将巡行状态划分为多个子状态。针对当前示例，Patrol 状态包含两个子状态，即移至当前路点并获取下一个路点。其中，路点表示为主体对象行进的目标位置。当实现这一类调整行为时，需要再次考察状态机。

首先，可点击画布空白区域并生成某一子状态，随后选取 Create Sub-State Machine。考虑到现有的原始 Patrol 状态以及全部连接，可将 Patrol 状态拖曳至新创建

的子状态中，并对二者进行合并。当 Patrol 状态拖曳至子状态上方时，将会出现一个+标识，这表示向另一个状态中添加了一个状态。当拖入 Patrol 状态时，新的子状态将吸入该状态。子状态包含不同的外观，即六边形状态而非矩形，如图 2.11 所示。随后可将该子状态命名为 Patrol。

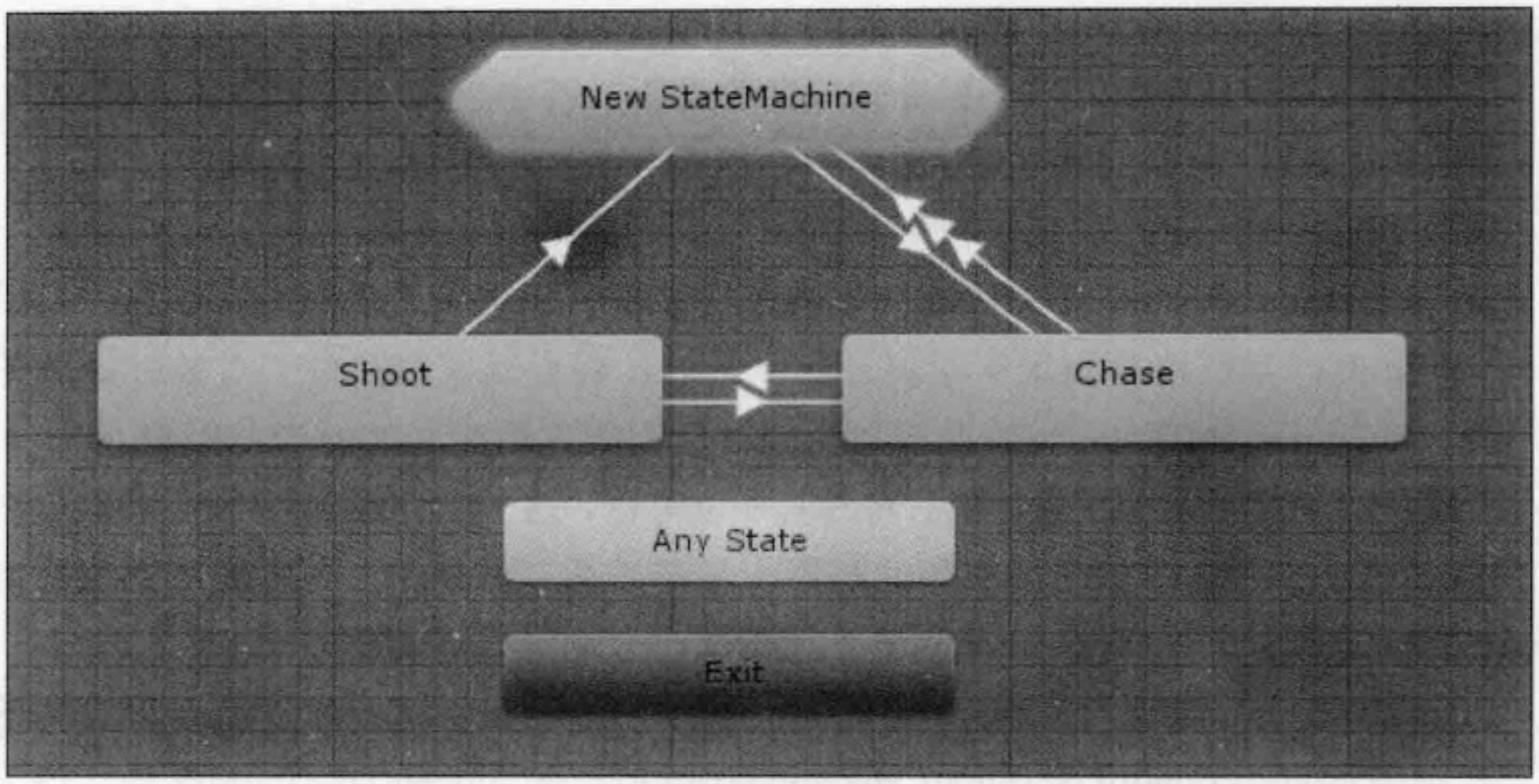

图 2.11

双击某一子状态即可进入该状态，并可视为从较低一级进入子状态。此处，读者应注意窗口中的某些状态变化。其中，Patrol 状态连接至称作“(Up) Base Layer”的节点，实际上表示为当前级别与状态机所处的上一级之间的连接。另外，Entry 状态则直接连接至 Patrol 状态。

然而，这并非是当前操作所需的结果——该过程表示为闭环，且无法从当前状态移至所需的路点状态，因而需要进行适当的调整。首先，可将子状态的名称修改为 PatrolEntry；其次，还需要进一步赋予某些转换。当进入 Entry 状态时，应确定是否需要持续移动至当前路点位置处，或者获取一个新的路点。对此，可将每一个结果表示为一个状态，因而需要创建两个状态，即 MovingToTarget 和 FindingNewTarget。随后，可在 PatrolEntry 与各新状态之间创建一个转换。类似地，还应在两个新状态间构建一个转换，即 MovingToTarget 状态与 FindingNewTarget 状态间的转换，反之亦然。此处，还需添加名为 distanceFromWaypoint 的 float 参数，并构建下列条件。

- PatrolEntry 状态转换至 MovingToTarget 状态：
  - distanceFromWaypoint > 1。
- PatrolEntry 状态转换至 FindingNewTarget 状态：
  - distanceFromWaypoint < 1。
- MovingToTarget 状态转换至 FindingNewTarget 状态：
  - distanceFromWaypoint < 1。

读者可能会产生如下疑问：为何不在新目标状态和 MovingToTarget 状态间赋予某一转换规则？其原因在于，此处将通过状态机行为执行代码，并于随后自动进入 MovingToTarget 状态，且无须附加任何条件。对此，可选择 FindingNewTarget 状态并加入某一行为，随后将其命名为 SelectWaypointState。

接下来，打开新脚本文件，移除除 OnStateEnter 之外的全部方法，并添加下列功能项：

```
TankAi tankAi = animator.gameObject.GetComponent<TankAi>();
tankAi.SetNextPoint();
```

当前操作旨在获取指向 TankAi 脚本的引用，并简单地调用其 SetNextPoint()方法。

最后，还需重新调整出射连接。当前新状态并未包含源自该级的转换，因而需要向“(Up) Base Layer”状态添加与 PatrolEntry 状态相同的条件。这也是 Any State 的作用之一，即支持任意状态间的转换，且无须考虑独立的转换条件。因此，此处不必在各状态与“(Up) Base Layer”状态间添加转换，且仅需向 Any State 添加一次即可。相应地，可在 Any State 与 PatrolEntry 之间添加某一转换，并针对“(Up) Base Layer”状态采用与 Entry 状态相同的条件。这可视为一种变通方案，且无法直接在 Any State 和“(Up) Base Layer”状态之间进行连接。

当上述操作结束后，对应的子状态机如图 2.12 所示。

4. 测试

当前任务是运行程序，并查看敌方坦克在提供的路点间的运动状态。如果将玩家对象置于敌方坦克的路径上，即可观察到动画器中的转换行为：从 Patrol 状态转至 Chase 状态；当玩家移出范围后，敌方坦克将再次返回至 Patrol 状态。需要注意的是，Chase 状态和 Shoot 状态并未完全实现，第 3 章将对此加以讨论。

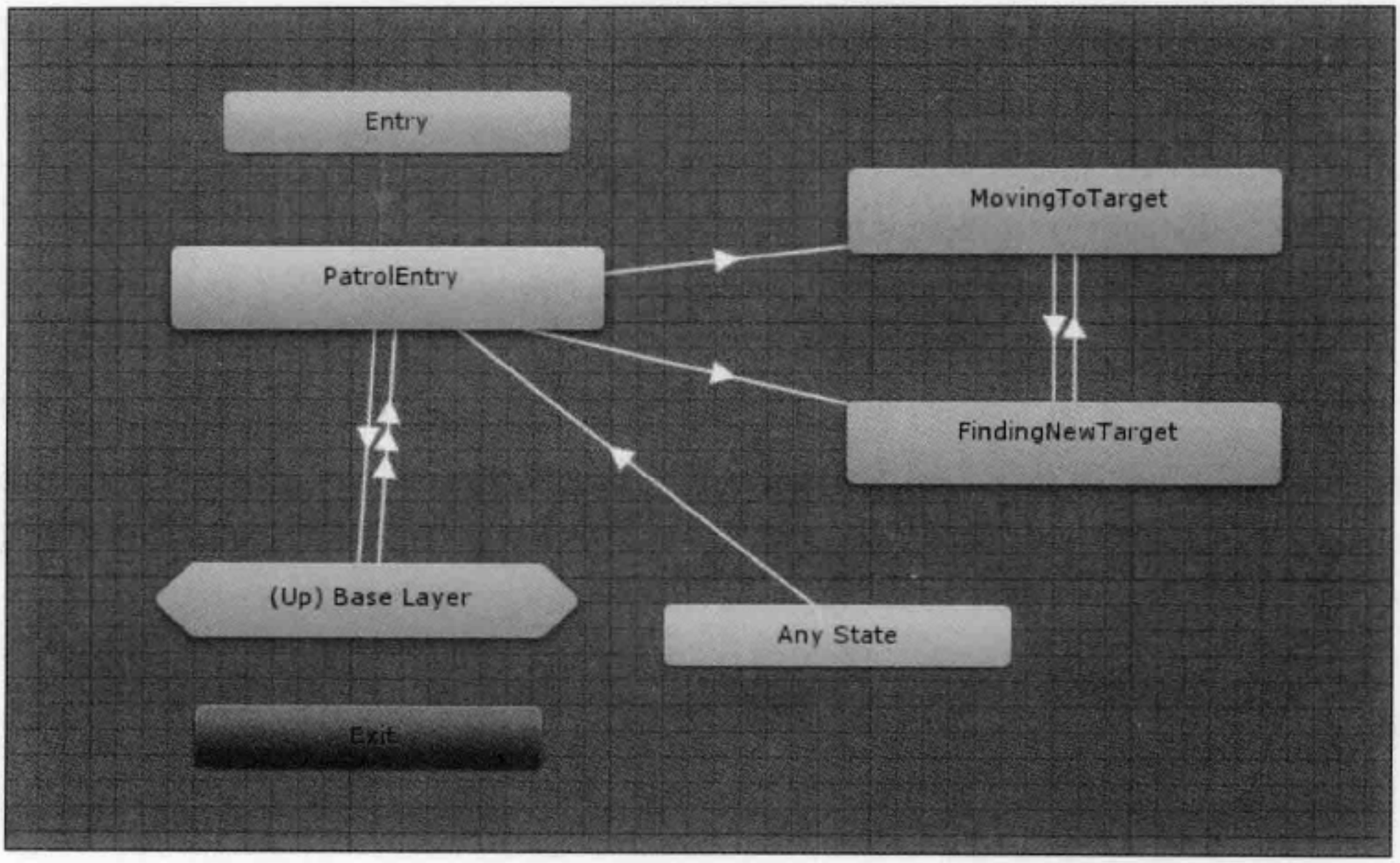

图 2.12

2.5 本 章 小 结

本章讨论了 Unity 5 中状态机的实现方式，并通过坦克游戏使用了基于动画控制器的状态机。除此之外，本章还介绍了状态间的状态机行为和转换。在此基础上，还将一个简单的状态机应用于主体对象上，进而创建了首个人工智能实体。

第 3 章将继续讨论坦克游戏，相对于主体对象的周边环境，并向其赋予更为复杂的感知方法。

第 3 章　实现感知系统

本章讨论 AI 的行为方式，并采用与生物体类似的感知系统。如前所述，角色的 AI 系统需要感知其周边环境，例如障碍物的位置、敌方角色的位置、敌方角色是否处于玩家的可见范围内等。NPC 的 AI 完全取决于其从周围环境获取的信息，根据此类信息，AI 角色可确定所执行的逻辑。如果针对 AI 缺乏足够的信息，AI 角色可能会呈现出奇怪的行为，例如选取错误的隐蔽地带、处于空闲状态、反复执行奇怪的动作，或者无法制定正确的决策。感兴趣的读者可在 YouTube 上搜索 AI 问题集锦，其中可看到 AI 角色有趣的行为，即使是 AAA 游戏也不乏此类错误。

对此，可检测全部环境参数以及预定义值。同时，使用适宜的设计模式有助于代码的维护，并可方便地对其进行扩展。本章将对传感系统的实现引入相应的设计模式，其中包括：

- 传感系统的具体内容。
- 现有的某些不同的传感系统。
- 如何利用传感机制构建简单的坦克对象。

3.1　基本的感知系统

AI 感知系统模拟视觉、听觉甚至是嗅觉系统，并以此跟踪或识别物体。在游戏 AI 传感系统中，主体对象根据目标周期性地检测周边环境，并通过感知系统进行判断。

基本的传感系统概念可描述为，该系统包含两个组件，即 Aspect 和 Sense 组件。感知系统将搜寻特定目标，例如敌方角色或罪犯。例如，巡逻士兵的感知系统可搜寻敌方游戏对象；或者，基于嗅觉感知的僵尸实体可搜索包含大脑的目标实体。

针对本章示例，这也是需要实现的基本内容，即称作 Scene 的基本接口，并通过其他自定义传感系统予以实现。另外，视觉系统可对周边环境进行观察，如果 AI 角色查看到敌人，则需要执行相应的动作。类似地，通过感知系统，角色还可对距离进行判断，甚至可“听到”附近的敌方角色。稍后将编写感知系统搜寻的简化

Aspect 类。

3.1.1　视锥

在第 2 章中提供的示例中，曾构建了主体对象并通过视线检测玩家坦克。其中，视线通过光线投射方式表示。光线投射可视为 Unity 的一个特性，并以此确定与直线（包含点和既定方向）相交的对象。尽管该方案可高效地处理视觉检测计算，但对于大多数实体而言，光线投射并未对视觉计算进行准确的建模。一种替代方法是采用视锥，如图 3.1 所示。其中，视域可通过锥体形状建模，并根据游戏类型实现为 2D 或 3D 锥体。

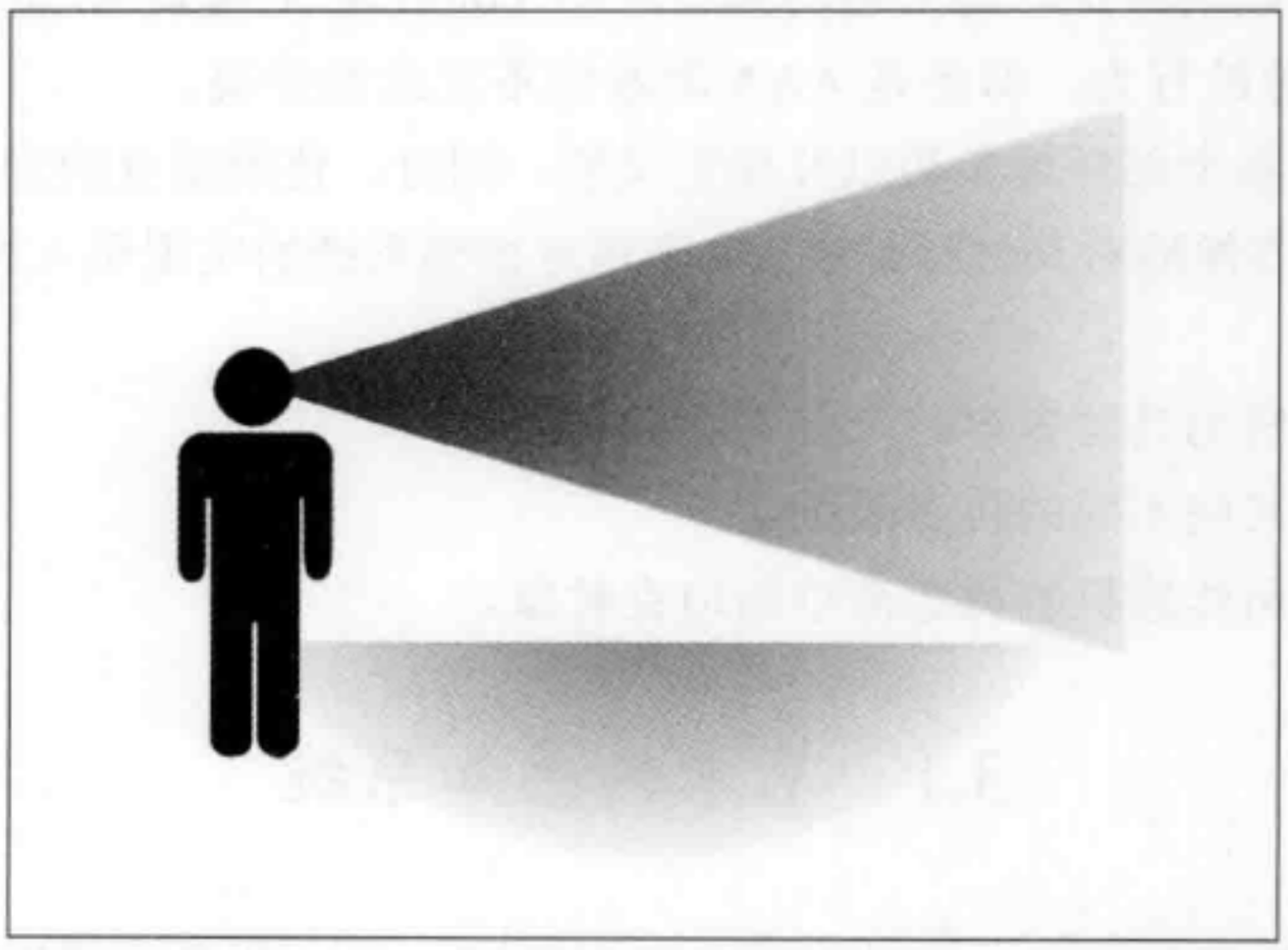

图 3.1

图 3.1 显示了视锥体这一概念。其中，视锥体始于主体对象的观察点（眼睛）；随着锥体不断延伸，其准确度将随着距离而降低，即图 3.1 中的褪色区域。

锥体的实现过程从基本的相交测试至复杂的真实模型（模拟眼睛视觉系统）不一而同。在简单的实现方案中，仅需要测试对象是否与视锥体相交即可，并忽略距离和边缘区域。相比较而言，复杂的实现方案则对眼睛的视见系统予以模拟。其中，当视锥体范围逐渐增大时，其视域也随之增加，与视线中间位置相比，对应的观察能力也有所降低。

3.1.2　基于球体的听觉、感觉和嗅觉

球体则是听觉、触觉以及嗅觉的一类简单、高效的建模方式。例如，对于听觉系统，可假设球心为源位置，随着距离的增加，音量（响度）则逐渐降低。相反，听者也可不采用音源建模（或者在此基础上建模）。此时，听觉系统可采用球体表示，且距离听者最近的声音将被听到。除此之外，还可相对于主体对象调整球体的尺寸和位置，并适应于触觉和嗅觉系统。

图 3.2 显示了球体以及主体对象与构建操作间的适配方式。

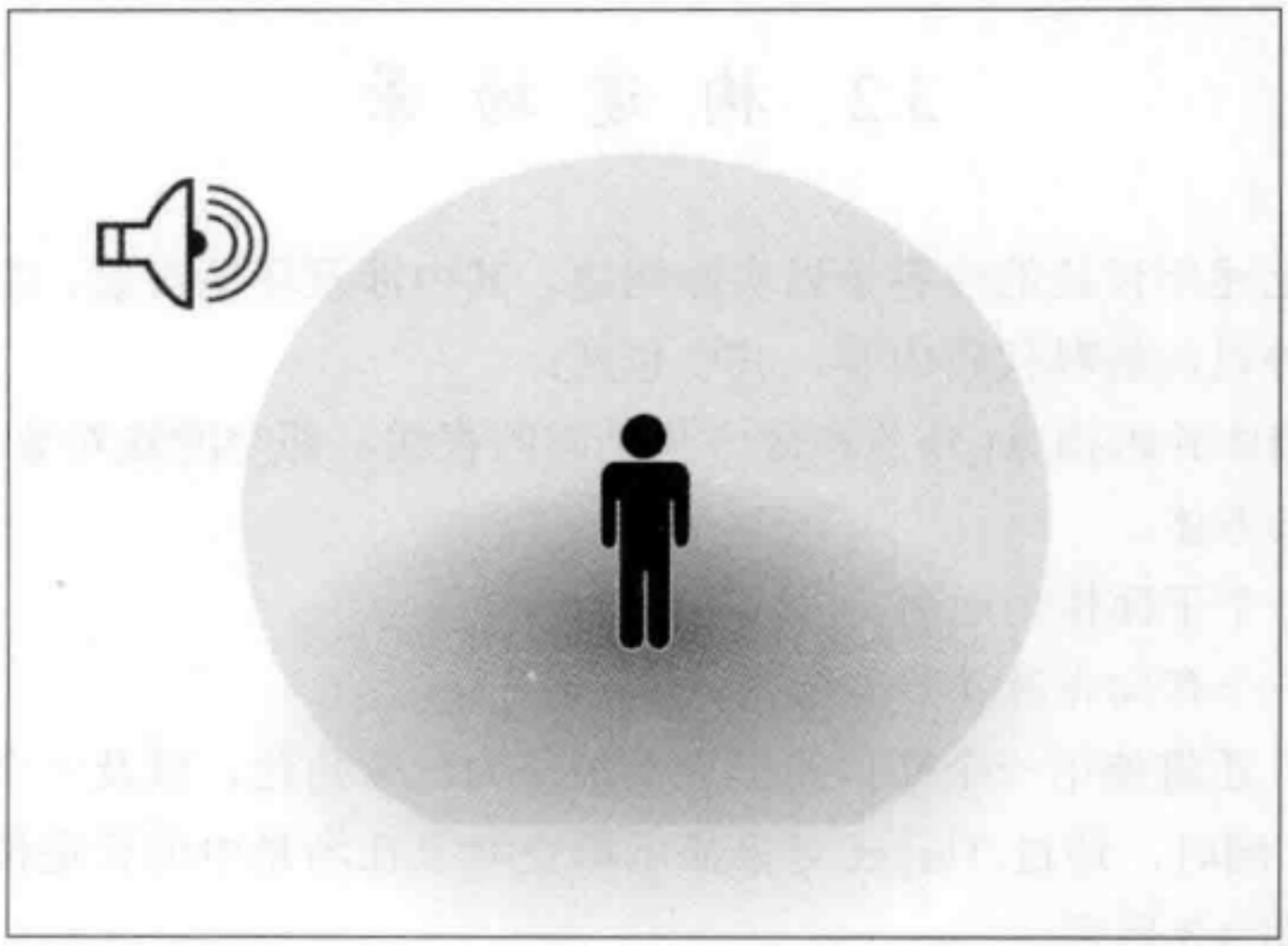

图 3.2

类似于视见系统，根据与感知系统间的距离，或者作为简单的相交事件，可对主体对象与感知事件间的概率进行调整。其中，当感知源与球体交叠时，即可检测到感知事件。

3.1.3　扩展 AI

主体对象可访问与周围环境相关的隐藏信息，或者游戏场景中的其他实体，这可视为一种强大的工具，但也带来了额外的复杂度。

在游戏中，可通过具体值对抽象概念进行建模。例如，可利用 0~100 内的数值表

示玩家的生命值。主体角色可访问此类信息并制定“逼真”的决策，而数据的访问过程并无太多新意。另外，主体角色还可在游戏中使用某种超自然力量或感知事件，且无须亲身体验实际场景。

3.1.4　感知系统的创新

上述内容阐述了主体对象感知周边环境的基本方式，但并非是唯一方式。若游戏需要使用到其他感知类型，则可对上述模式进行适当整合。例如，可采用圆柱体或球体表示视域，或者利用盒体表示嗅觉感知系统。

3.2　构 建 场 景

下面将对上述所讨论的内容予以实际构建，其中涉及环境对象、主体对象以及其他相关对象，并以此解释代码内容，其中包括：

- 创建墙体并阻挡 AI 角色和敌方角色间的视线，即空游戏对象 Obstacles 下的多个长方体。
- 添加一个平面作为地面。
- 添加一个有向光源并照亮场景。

除此之外，还将使用一个简单的坦克模型作为玩家角色，以及一个简单的立方体作为 AI 角色。同时，通过 Target 对象显示坦克对象在场景中的行进位置。最终的场景层次结构如图 3.3 所示。

图 3.3

下面将在场景中随机定位坦克对象、AI 角色以及墙体，并适当增加平面的尺寸。在该示例中，所有对象均不会落入平面下方。另外，还应对相机进行适当调整，并呈现如图 3.4 所示的清晰画面。

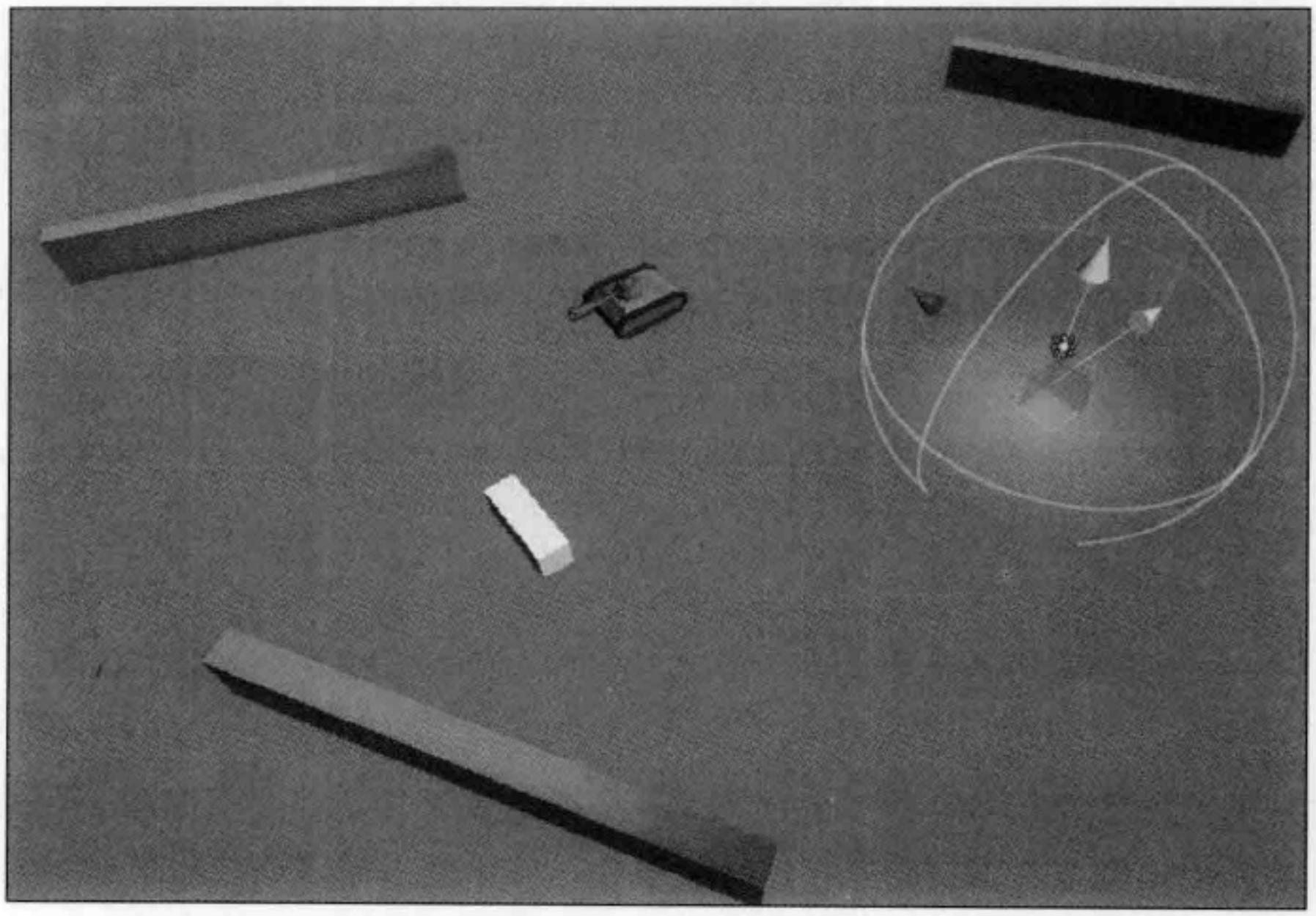

图 3.4

待基本环境构建完毕后，下面考察坦克对象、AI 角色以及玩家角色的实现过程。

3.3　创建玩家坦克

Target 对象定义为简单的球体对象（且禁用网格渲染）。同时，还将创建一个点光源，并使其表示为 Target 对象的子对象。此处应确保光源位于中心位置，进而有效地发挥其应有的作用。

Target.cs 文件如下所示：

```
using UnityEngine;
using System.Collections;
```

```
public class Target : MonoBehaviour {

  public Transform targetMarker;

  void Update () {
    int button = 0;
    //Get the point of the hit position when the mouse is being
//clicked.
    if (Input.GetMouseButtonDown(button)) {
      Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
      RaycastHit hitInfo;
      if (Physics.Raycast(ray.origin, ray.direction, out hitInfo)) {
        Vector3 targetPosition = hitInfo.point;
        targetMarker.position = targetPosition;
      }
    }
  }
}
```

此处需要将该脚本绑定至 Target 对象上，在查看器中，可将其赋予 targetMarker 变量。相应地，脚本负责检测鼠标的单击事件，并于随后通过光线投射技术检测 3D 空间内平面上的鼠标点击点，进而将 Target 对象更新至场景中的该点处。

3.3.1　实现玩家坦克

玩家坦克表示为第 2 章所用的简单坦克模型，绑定了非运动刚体组件，并在与 AI 角色间执行碰撞检测时生成触发事件。对此，可将标签 Player 赋予当前坦克对象。

坦克对象通过 PlayerTank 脚本进行控制，稍后将对其加以实现。该脚本接收地图上的目标位置，并更新其目标点和方向。

PlayerTank.cs 文件中的代码如下所示：

```
using UnityEngine;
using System.Collections;

public class PlayerTank : MonoBehaviour {
```

```
public Transform targetTransform;
private float movementSpeed, rotSpeed;

void Start () {
  movementSpeed = 10.0f;
  rotSpeed = 2.0f;
}

void Update () {
  //Stop once you reached near the target position
    if (Vector3.Distance(transform.position,
      targetTransform.position) < 5.0f)
      return;

    //Calculate direction vector from current position to target
//position
    Vector3 tarPos = targetTransform.position;
    tarPos.y = transform.position.y;
    Vector3 dirRot = tarPos - transform.position;

    //Build a Quaternion for this new rotation vector
    //using LookRotation method
    Quaternion tarRot = Quaternion.LookRotation(dirRot);

    //Move and rotate with interpolation
      transform.rotation= Quaternion.Slerp(transform.rotation,
         tarRot, rotSpeed * Time.deltaTime);

      transform.Translate(new Vector3(0, 0,
         movementSpeed * Time.deltaTime));
    }
}
```

待脚本应用于坦克对象上后，图3.5显示了查看器中的脚本内容。

脚本接收地图上的目标位置，并更新其目标点和方向。当脚本赋予坦克对象后，应确保Target对象赋予targetTransform变量。

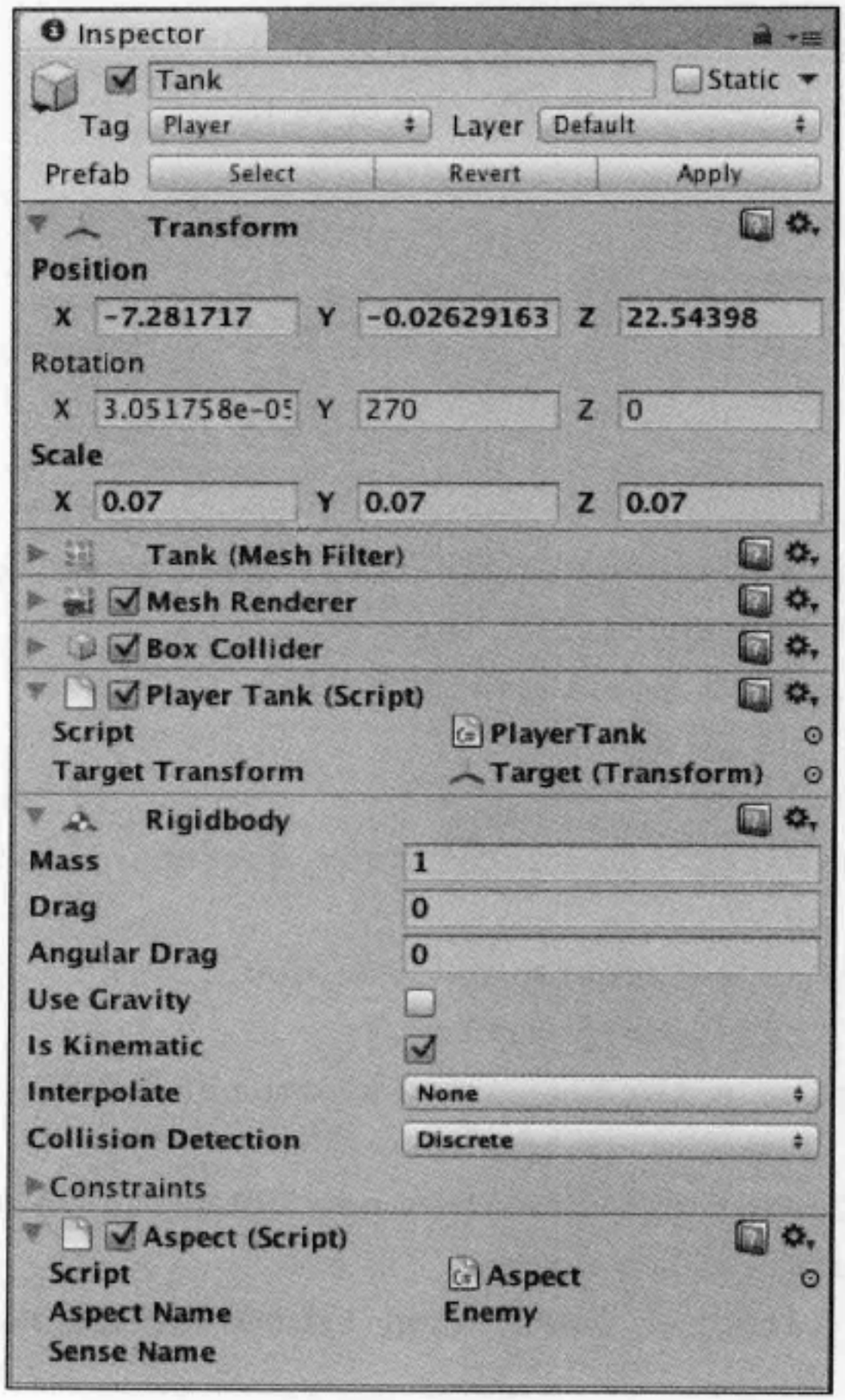

图 3.5

3.3.2　实现 Aspect 类

下面考察 Aspect.cs 类，该类相对简单，且仅包含了一个名为 aspectName 的公有属性，这也是本章所需的全部变量。当 AI 角色对其周边环境有所感知时，将通过 aspectName 对其进行检测，进而查看是否为 AI 角色所搜寻的目标。

Aspect.cs 文件如下所示：

```
using UnityEngine;
using System.Collections;
```

```
public class Aspect : MonoBehaviour {
  public enum aspect {
    Player,
    Enemy
  }
  public aspect aspectName;
}
```

此处可将该脚本与玩家坦克对象进行绑定，并将 aspectName 属性设置为 Enemy，如图 3.6 所示。

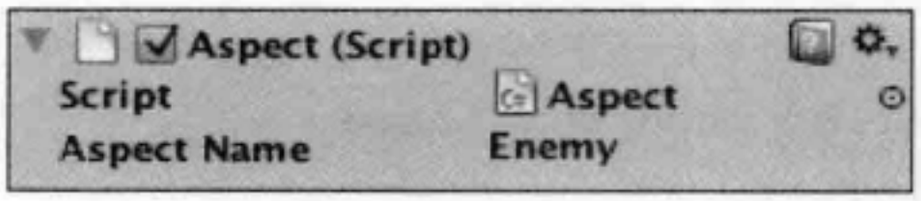

图 3.6

3.4　创建 AI 角色

AI 角色将以随机方向在场景中漫游，同时包含下列两种感知系统：

- 视见系统查看敌方角色是否位于可见范围以及有效距离内。
- 触觉系统检测敌方角色是否与盒体间产生碰撞，并包围当前 AI 角色。

如前所述，玩家坦克包含了 Enemy 对象，因而当玩家坦克被检测到时，上述感知系统也将被触发。

Wander.cs 文件如下所示：

```
using UnityEngine;
using System.Collections;

public class Wander : MonoBehaviour {
  private Vector3 tarPos;

  private float movementSpeed = 5.0f;
  private float rotSpeed = 2.0f;
  private float minX, maxX, minZ, maxZ;

  //Use this for initialization
```

```
    void Start () {
      minX = -45.0f;
      maxX = 45.0f;

      minZ = -45.0f;
      maxZ = 45.0f;

      //Get Wander Position
      GetNextPosition();
    }

    //Update is called once per frame
    void Update () {
      //Check if we're near the destination position
      if (Vector3.Distance(tarPos, transform.position) <= 5.0f)
        GetNextPosition(); //generate new random position

      //Set up quaternion for rotation toward destination
      Quaternion tarRot = Quaternion.LookRotation(tarPos -
          transform.position);

      //Update rotation and translation
      transform.rotation = Quaternion.Slerp(transform.rotation,
          tarRot,rotSpeed * Time.deltaTime);

      transform.Translate(new Vector3(0, 0,
          movementSpeed * Time.deltaTime));
    }

    void GetNextPosition() {
      tarPos = new Vector3(Random.Range(minX, maxX), 0.5f,
          Random.Range(minZ, maxZ));
    }
}
```

当 AI 角色到达当前目标位置时，Wander 脚本将生成新的随机位置，随后，Update

方法将旋转敌方角色，并使其朝向新的目标行进。此处，需要将上述脚本绑定至 AI 角色上，从而使得该角色可在场景中行进。

3.5　使用 Sense 类

Sense 类定义为感知系统的接口，并通过其他自定义感知系统予以实现。该类定义了两个虚方法，即 Initialize 和 UpdateSense 方法。此类方法在自定义感知系统中实现，并在 Start 和 Update 方法中予以执行。

Sense.cs 文件如下所示：

```
using UnityEngine;
using System.Collections;

public class Sense : MonoBehaviour {
  public bool bDebug = true;
  public Aspect.aspect aspectName = Aspect.aspect.Enemy;
  public float detectionRate = 1.0f;

  protected float elapsedTime = 0.0f;

  protected virtual void Initialize() { }
  protected virtual void UpdateSense() { }

  //Use this for initialization
  void Start () {
    elapsedTime = 0.0f;
    Initialize();
  }

  //Update is called once per frame
  void Update () {
    UpdateSense();
  }
}
```

该类的基本属性包括执行感知操作的检测率，以及搜寻的对应目标。需要注意的是，该脚本不应与任何对象进行绑定。

3.6 视 见 功 能

视见感知负责检测特定目标是否位于视域以及可见范围内。当发现目标时，该系统将执行特定的操作行为。

Perspective.cs 文件如下所示：

```
using UnityEngine;
using System.Collections;

public class Perspective : Sense {
  public int FieldOfView = 45;
  public int ViewDistance = 100;

  private Transform playerTrans;
  private Vector3 rayDirection;

  protected override void Initialize() {

    //Find player position
    playerTrans =
GameObject.FindGameObjectWithTag("Player").transform;
}

//Update is called once per frame
protected override void UpdateSense() {
  elapsedTime += Time.deltaTime;

  //Detect perspective sense if within the detection rate
  if (elapsedTime >= detectionRate) DetectAspect();
}
```

```
//Detect perspective field of view for the AI Character
void DetectAspect() {

RaycastHit hit;

//Direction from current position to player position
rayDirection = playerTrans.position -
    transform.position;

//Check the angle between the AI character's forward
  //vector and the direction vector between player and AI
  if ((Vector3.Angle(rayDirection, transform.forward)) <
FieldOfView) {
    //Detect if player is within the field of view
    if (Physics.Raycast(transform.position, rayDirection,
        out hit, ViewDistance)) {
      Aspect aspect =
      hit.collider.GetComponent<Aspect>();

      if (aspect != null) {
        //Check the aspect
        if (aspect.aspectName == aspectName) {
          print("Enemy Detected");
        }
      }
    }
  }
}
```

此处需要实现 Initialize 和 UpdateSense 方法，并在父 Sense 类的 Start 和 Update 方法中被调用。随后，DetectAspect 方法首先检测玩家和 AI 当前方向间的夹角。如果位于视见范围内，可向玩家坦克所处位置投射一条光线，该光线的长度表示为可见距离值。当首次与其他对象碰撞时，Raycast 方法返回。除此之外，还需检测目标组件及其名称。通过这种方式，即使玩家位于可见范围内，AI 角色仍然无法看到隐藏于墙后的玩家。

OnDrawGizmos 方法根据视域和视见距离绘制直线，进而在测试过程中可在编辑器窗口中查看到 AI 角色的视线。相应地，可将该脚本绑定于 AI 角色上，并确保目标名称为 Enemy。

上述方法的具体内容如下所示：

```
void OnDrawGizmos() {
  if (playerTrans == null) return;

  Debug.DrawLine(transform.position, playerTrans.position, Color.
red);

    Vector3 frontRayPoint = transform.position +
       (transform.forward * ViewDistance);

    //Approximate perspective visualization
    Vector3 leftRayPoint = frontRayPoint;
    leftRayPoint.x += FieldOfView * 0.5f;

    Vector3 rightRayPoint = frontRayPoint;
    rightRayPoint.x -= FieldOfView * 0.5f;

    Debug.DrawLine(transform.position, frontRayPoint, Color.green);

    Debug.DrawLine(transform.position, leftRayPoint, Color.green);

      Debug.DrawLine(transform.position, rightRayPoint, Color.green);
  }
}
```

3.7 触 觉 系 统

另一个需要实现的感知系统则是 Touch.cs，当玩家实体位于附近 AI 实体的特定区域内时，该系统将被触发。当前 AI 角色包含了盒体碰撞组件，其 IsTrigger 标记处于启用状态。

当组件间产生碰撞时，需要实现所触发的 OnTriggerEnter 事件。考虑到坦克实体包含了碰撞体和刚体组件，因而当 AI 角色和玩家坦克碰撞体之间产生碰撞时，将会引发碰撞事件。

Touch.cs 文件中的代码如下所示：

```
using UnityEngine;
using System.Collections;

public class Touch : Sense {
  void OnTriggerEnter(Collider other) {
    Aspect aspect = other.GetComponent<Aspect>();

    if (aspect != null) {
      //Check the aspect
      if (aspect.aspectName == aspectName) {
        print("Enemy Touch Detected");
      }
    }
  }
}
```

图 3.7 显示了上述方法中所涉及的触发器。

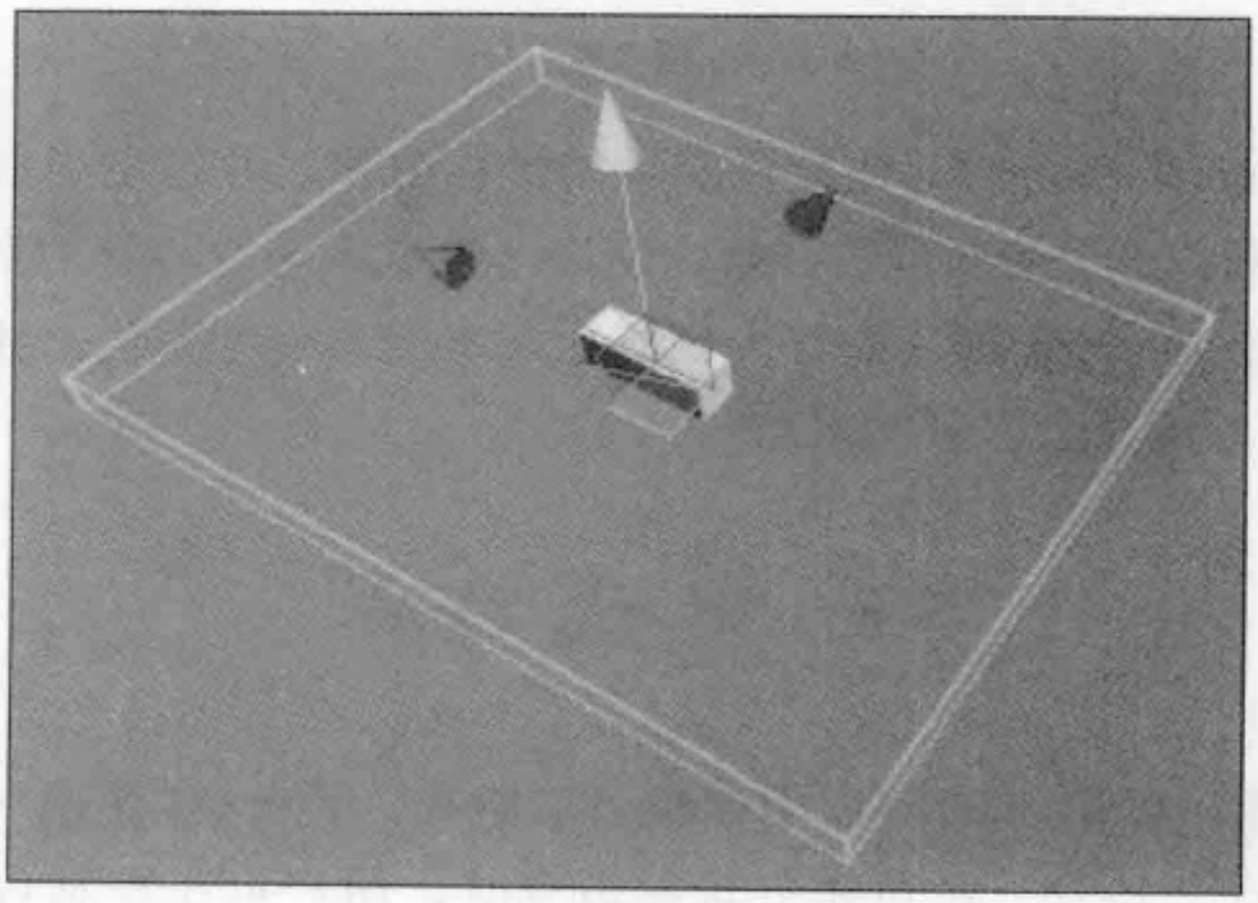

图 3.7

具体而言，图 3.7 显示了用于实现触觉系统的、敌方 AI 角色的盒体碰撞体。图 3.8 则显示了 AI 角色的构建过程。

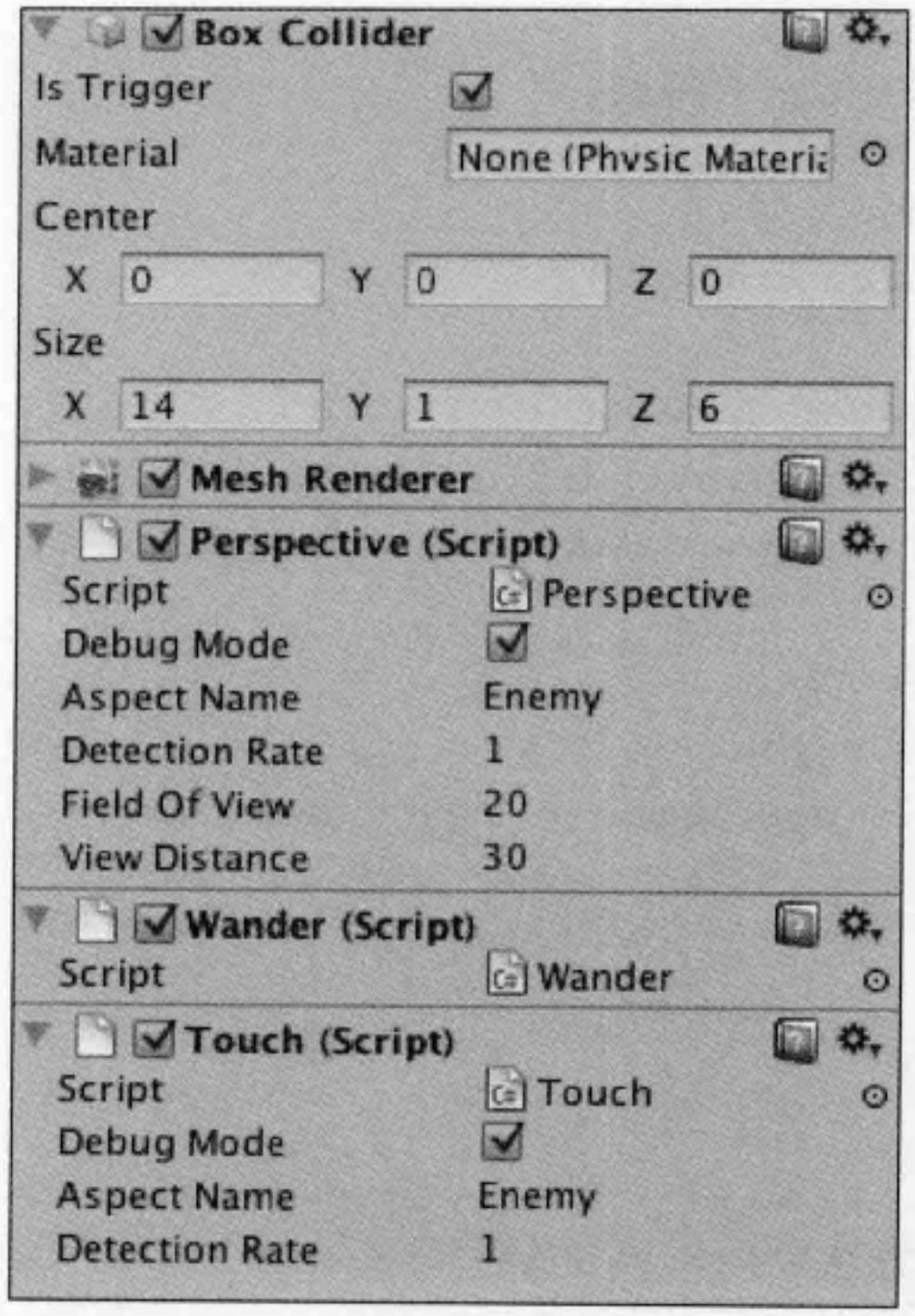

图 3.8

OnTriggerEnter 方法访问其他配置实体的目标组件，并检测当前目标名称是否为 AI 角色搜寻的目标。出于演示目的，此处仅输出触觉系统检测到的敌方目标名称。在实际项目中，还可进一步实现其他行为，例如玩家朝向敌方角色，并执行追逐、攻击等行为。

3.8　测 试 结 果

通过点击地面，可在 Unity 3D 中体验当前游戏，并在 AI 角色（处于巡视状态）附近移动玩家坦克。当 AI 角色靠近玩家坦克时，用户可在控制台日志窗口中看到“Enemy touch detected”消息。

图 3.9 显示了包含触觉和视觉系统的 AI 主体对象，并搜索敌方目标。若用户将玩家坦克移至 AI 前方，则会显示“Enemy detected”消息。如果在游戏运行过程中查看编辑器窗口，则会查看到调试绘制过程，其原因在于，OnDrawGizmos 方法在 Scene 类中加以实现。

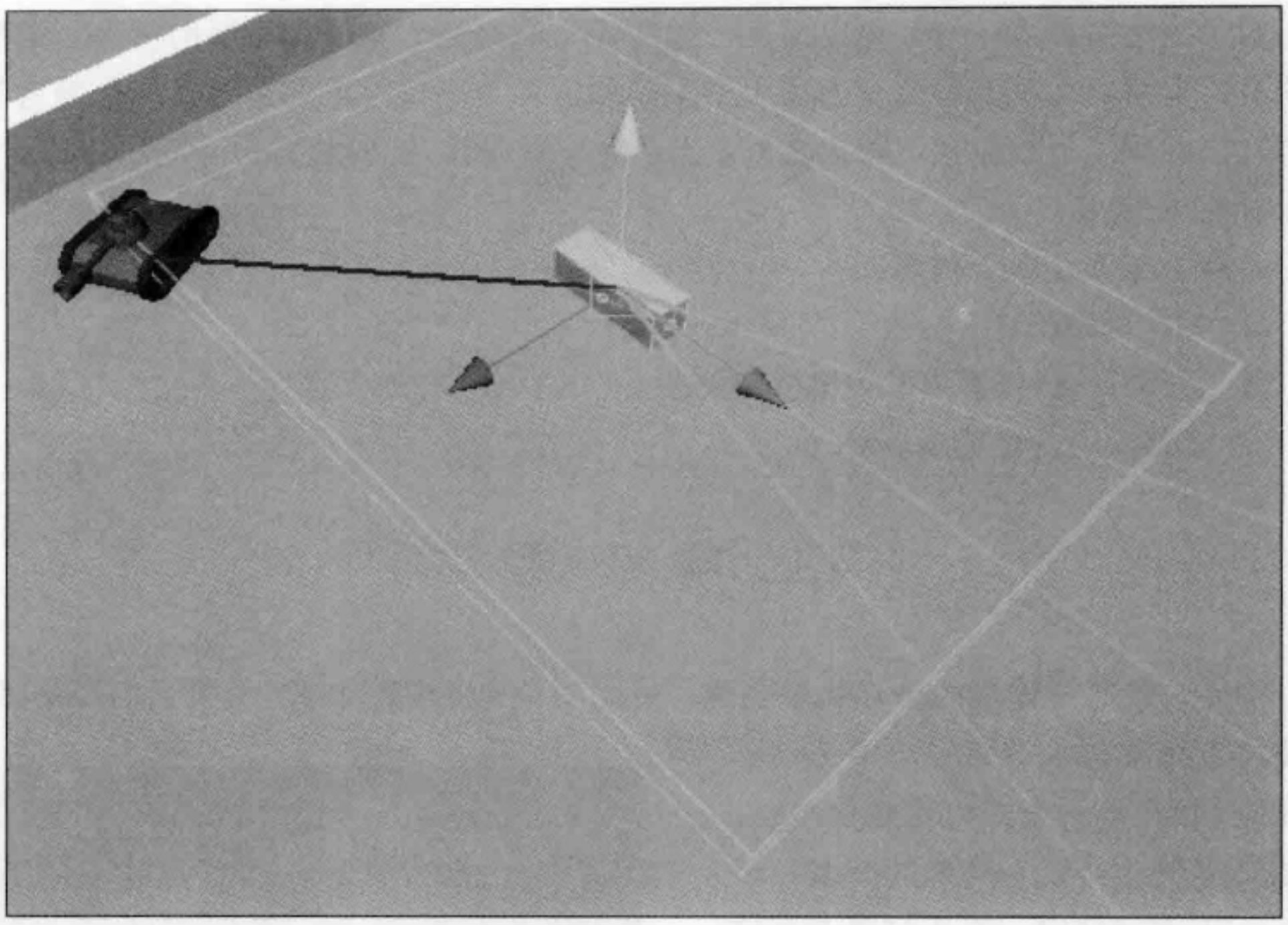

图 3.9

3.9　本 章 小 结

针对游戏 AI，本章介绍了感知系统，并实现了 AI 角色视觉和触觉系统。其中，感知系统可视为整体 AI 系统中决策系统中的部分内容，并可结合行为树针对特定感知检测某种行为。例如，当检测到敌方角色位于视线范围内，可利用 FSM 将 Patrol 状态调整为 Chase 状态或 Attack 状态。第 6 章将对行为树应用加以讨论。

第 4 章将考察 Unity3D 中的群集实现方法，以及 Craig Reynold 群集算法。

第4章 寻路方案

对于到达目标点的 AI 角色而言，躲避障碍物则是一类较为简单的行为。需要注意的是，本章所实现的特定行为用于群集模拟，各个主体对象的主要目标是躲避其他主体对象并到达目标位置。除此之外，当前方案并未涉及高效且最短路径，对此，4.2 节将讨论 A*算法。

本章主要涉及以下内容：

- 路径跟踪和转向。
- 自定义 A*寻路算法及其实现方案。
- Unity 中内建的 NavMesh。

4.1 路径跟踪

路径一般通过连接的路点加以创建，对此，本节将构建如图 4.1 所示的简单路径，并使立方体实体沿该路径平滑运动。

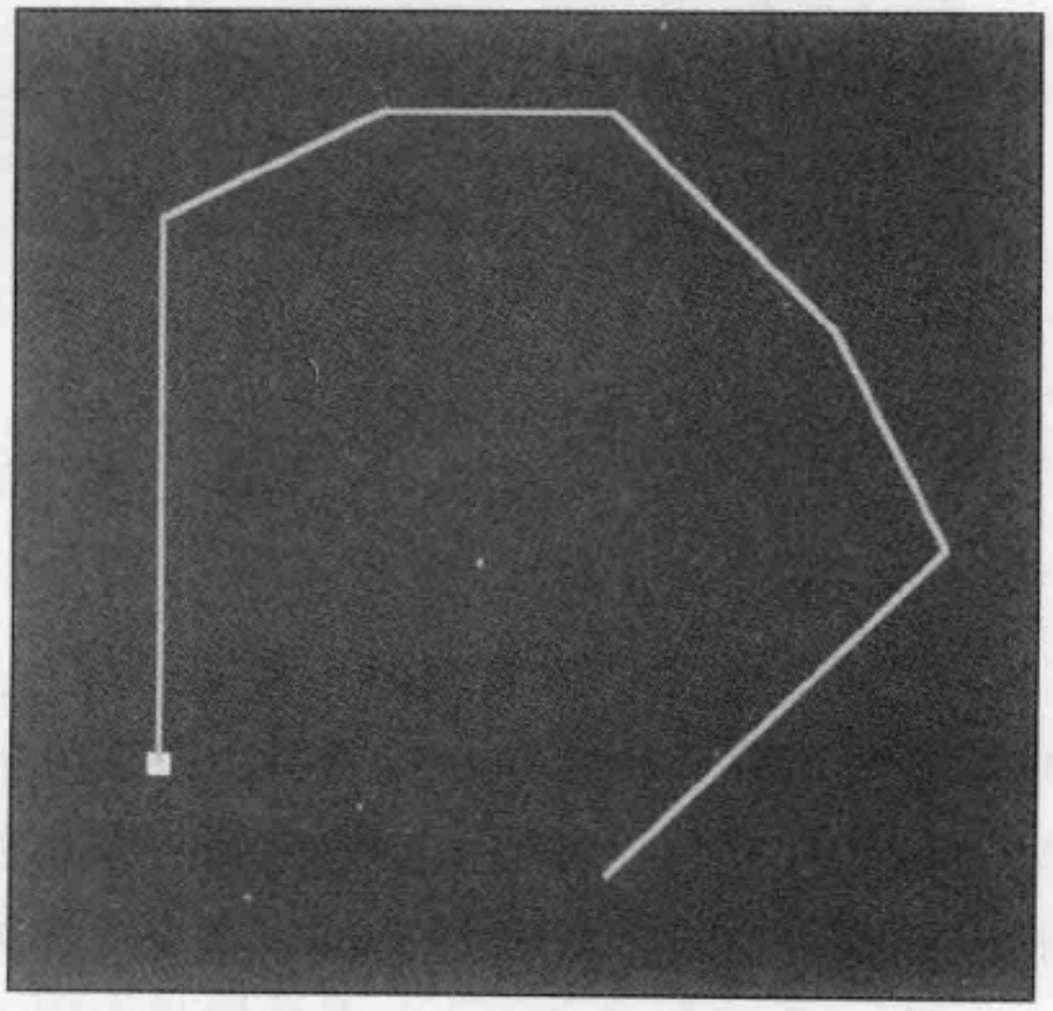

图 4.1

相应地，存在多种方式可构建该路径，此处仅实现一类较为简单的方案。在 Path.cs 脚本文件中，路点位置将存储于数组 Vector3 中，并于随后在编辑器中手工输入位置信息，该过程相对枯燥。一种可选方案是将空游戏对象的位置用作路点；或者可创建自己的编辑器插件，并实现此类任务的自动化操作，但这超出了本书的讨论范围。当前，考虑到路点数量相对较少，因而手工输入路点信息即可。

首先，可创建空游戏实体，并添

加路径脚本组件，如图 4.2 所示。

随后将显示包含全部路径路点的 Point A 变量，如图 4.3 所示。

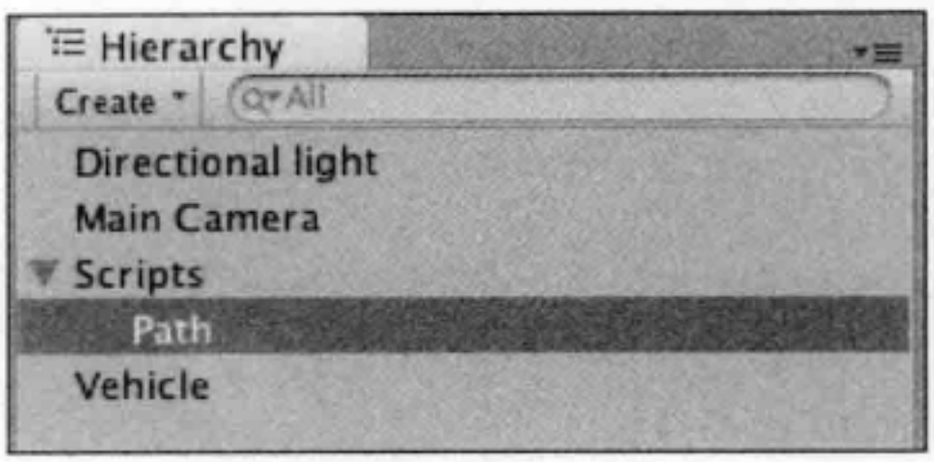

图 4.2

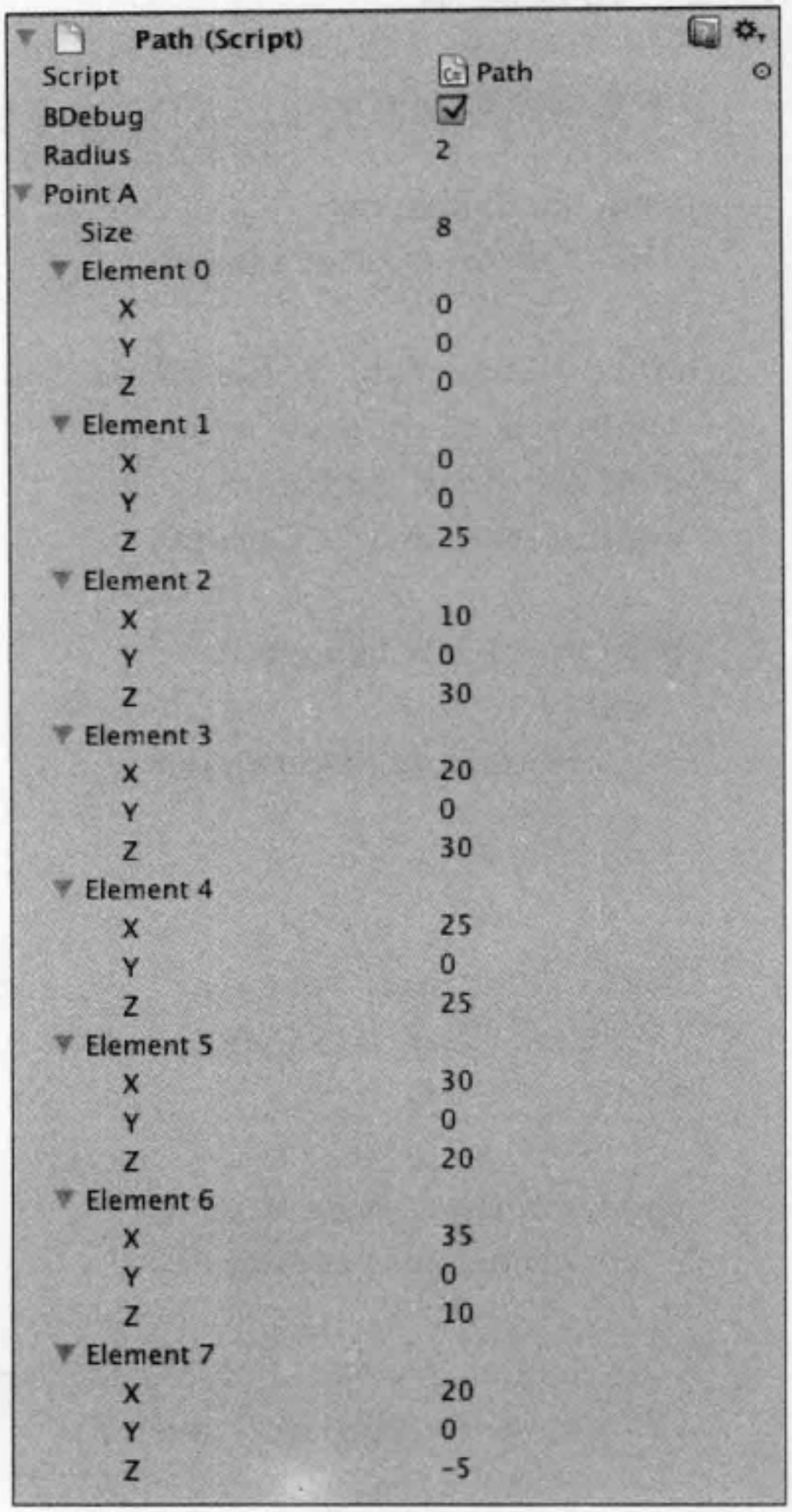

图 4.3

图 4.3 显示了创建路径所需的全部路点，其他两项属性分别是 debug mode 和 radius。若选择了 debug 模式，输入位置形成的路径在编辑器窗口中将采用线框加以绘制。radius 属性对于路径跟踪实体而言表示为一个范围值，当位于该半径范围内时，可知晓何时到达特定的路点。考虑到到达精确位置通常较为困难，因而这一范围半径

针对路径跟踪主体对象提供了一种高效方案。

4.1.1　路径脚本

下面考察路径脚本文件，该文件负责管理对象的路径。Path.cs 文件如下所示：

```
using UnityEngine;
using System.Collections;

public class Path : MonoBehaviour {
  public bool bDebug = true;
  public float Radius = 2.0f;
  public Vector3[] pointA;

  public float Length {
    get {
      return pointA.Length;
    }
  }

  public Vector3 GetPoint(int index) {
    return pointA[index];
  }

  void OnDrawGizmos() {
    if (!bDebug) return;

    for (int i = 0; i <pointA.Length; i++) {
      if (i + 1<pointA.Length) {
        Debug.DrawLine(pointA[i], pointA[i + 1],
          Color.red);
      }
    }
  }
}
```

不难发现，上述脚本文件较为简单，其中包含了 Length 属性，并返回路点数组的

长度和尺寸。GetPoint 方法返回既定数组索引处特定路点的 Vector3 位置。随后，Unity 帧调用 OnDrawGizmos 方法，并在编辑器环境下绘制组件。如果未开启游戏视图右上方的线框，则当前绘制结果将无法在游戏视图窗口中进行渲染。

4.1.2 使用路径跟踪器

当前示例中还包含了车辆实体，即简单的立方体对象，稍后将采用 3D 模型替换这些立方体对象。待脚本文件编写完毕后，即可添加 VehicleFollowing 脚本组件，如图 4.4 所示。

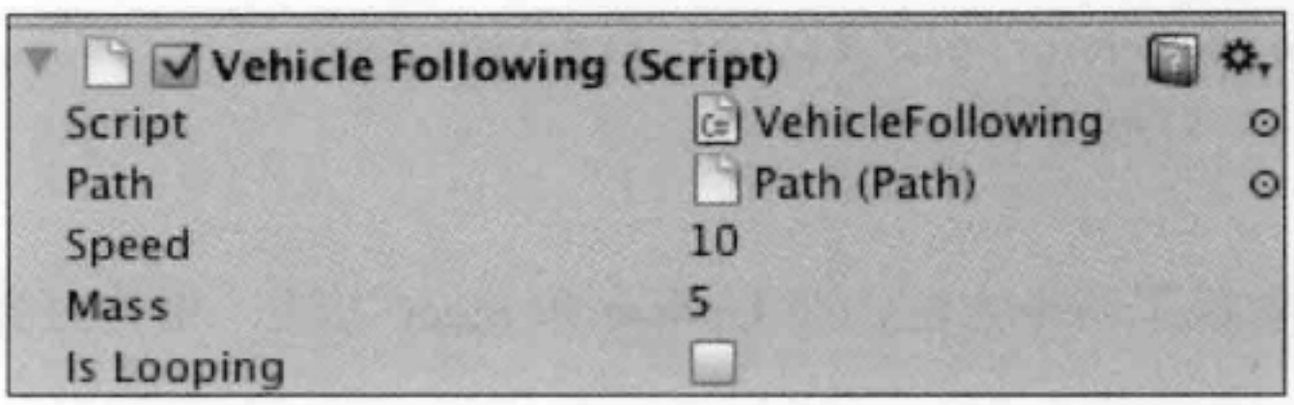

图 4.4

上述脚本使用了多个参数，首先是路径跟踪对象的引用，随后是 Speed 和 Mass 属性，进而计算加速度属性。另外，如果 IsLooping 标记被选中，则实体将以连续方式跟踪路径。VehicleFollowing.cs 文件如下所示：

```
using UnityEngine;
using System.Collections;

public class VehicleFollowing : MonoBehaviour {
  public Path path;
  public float speed = 20.0f;
  public float mass = 5.0f;
  public bool isLooping = true;

  //Actual speed of the vehicle
  private float curSpeed;

  private int curPathIndex;
  private float pathLength;
```

```
    private Vector3 targetPoint;

    Vector3 velocity;
```

首先将初始化相关属性，并利用 Start 方法中实体的前向向量设置速度向量的方向，对应代码如下所示：

```
void Start () {
    pathLength = path.Length;
    curPathIndex = 0;

    //get the current velocity of the vehicle
    velocity = transform.forward;
}
```

上述脚本仅实现了两个方法，即 Update 和 Steer 方法，相关内容如下所示：

```
void Update () {
    //Unify the speed
    curSpeed = speed * Time.deltaTime;

    targetPoint = path.GetPoint(curPathIndex);

    //If reach the radius within the path then move to next
      //point in the path
        if (Vector3.Distance(transform.position, targetPoint) <
          path.Radius) {
        //Don't move the vehicle if path is finished
       if (curPathIndex < pathLength - 1) curPathIndex++;
        else if (isLooping) curPathIndex = 0;
        else return;
}

//Move the vehicle until the end point is reached in
    //the path
    if (curPathIndex >= pathLength ) return;

//Calculate the next Velocity towards the path
```

```
    if (curPathIndex >= pathLength-1&& !isLooping)
        velocity += Steer(targetPoint, true);
        else velocity += Steer(targetPoint);

  //Move the vehicle according to the velocity
    transform.position += velocity;
  //Rotate the vehicle towards the desired Velocity
    transform.rotation = Quaternion.LookRotation(velocity);
}
```

在 Update 方法中，将检测实体是否已到达特定路点，即计算当前位置和路径半径范围间的距离。如果该实体位于当前范围内，只需增加索引值，并在路点数组中进行查找。如果对应路点为最后一个路点，则检测 isLooping 标记是否已被设置。若已设置，则可将当前目标设置为起始路点；否则，仅需在该点处停止即可。代码的下一部分内容将根据 Start 方法计算加速度。随后，将旋转实体对象，并根据速度值和方向对位置进行更新，如下所示：

```
//Steering algorithm to steer the vector towards the target
  public Vector3 Steer(Vector3 target,
    bool bFinalPoint = false) {
  //Calculate the directional vector from the current
    //position towards the target point
  Vector3 desiredVelocity = (target -transform.position);
  float dist = desiredVelocity.magnitude;

  //Normalise the desired Velocity
  desiredVelocity.Normalize();

  //Calculate the velocity according to the speed
  if (bFinalPoint&&dist<10.0f) desiredVelocity *=
    (curSpeed * (dist / 10.0f));
    else desiredVelocity *= curSpeed;

  //Calculate the force Vector
  Vector3 steeringForce = desiredVelocity - velocity;
  Vector3 acceleration = steeringForce / mass;
```

```
        return acceleration;
      }
    }
```

Steer 方法接收目标位置参数，该参数表示为路径中最后一个路点的 Vector3 位置。对此，首先需要计算当前位置至目标位置间的剩余距离。目标位置向量减去当前位置向量将生成指向目标位置的向量，该向量的量值表示为剩余距离，随后，可对该向量执行规范化操作，并保留其方向属性。如果当前路点为最后一个路点，且距离值小于 10，则可根据至目标点的剩余距离逐渐降低速度，直至速度值最终变为 0。否则，需要通过特定速度值更新目标速度。当从该目标速度向量中减去当前速度向量，即可计算新的转向向量。随后，利用该向量除以实体质量即可获得加速度。

当运行上述场景时，可以看到立方体将沿既定路径运动。同时，用户还可在编辑器视图窗口中查看绘制的路径。读者可尝试调整速度、质量和半径，并查看对系统整体行为所产生的影响。

4.1.3　躲避障碍物

本节将构建如图 4.5 所示的场景，AI 实体在到达目标点时将对障碍物予以躲避。此处所用的算法基于光线投射方案且相对简单，仅可躲避其前方的路径障碍物。

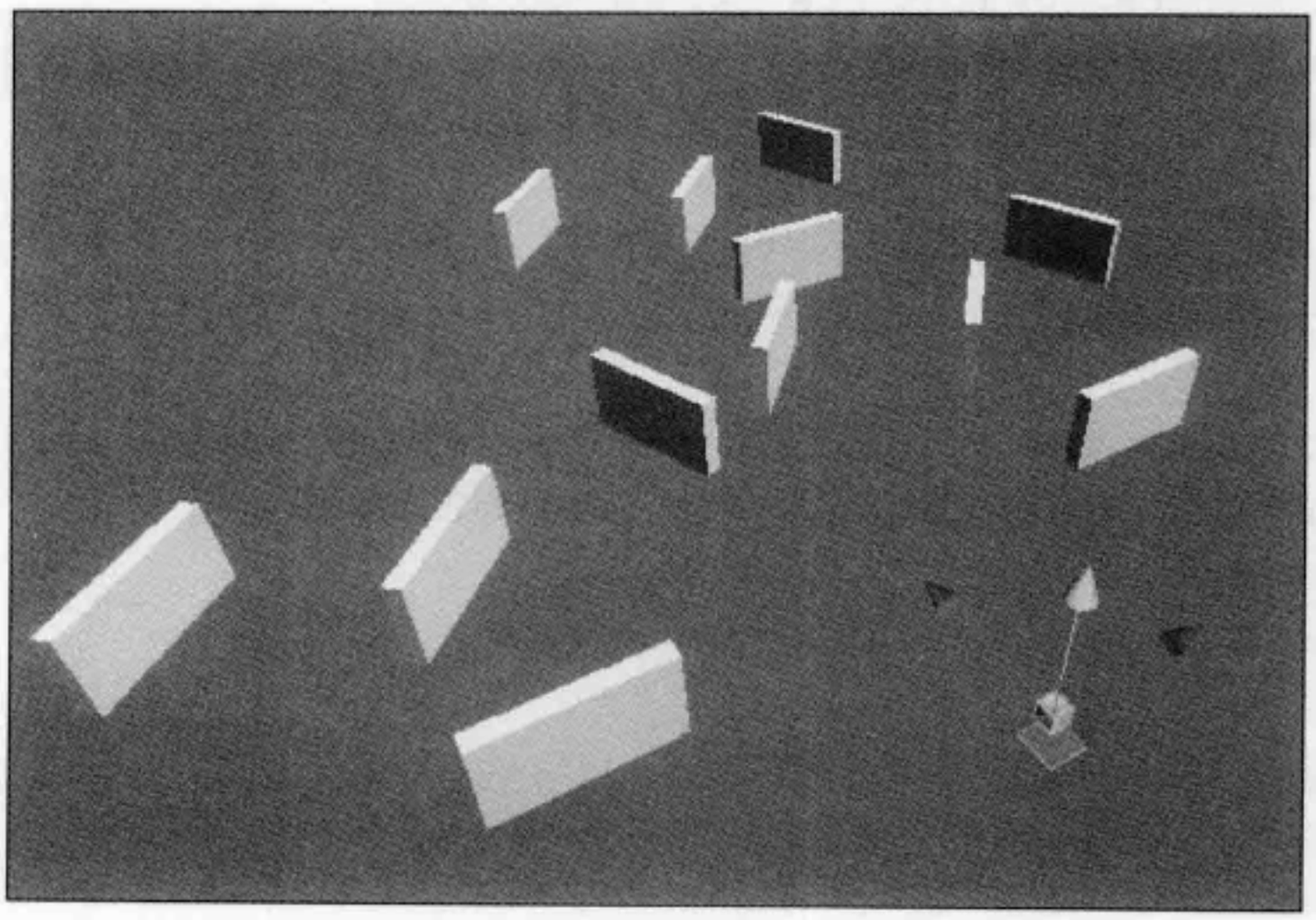

图 4.5

当创建上述场景时，可生成多个立方体实体，并在 Obstacles 空游戏对象下对其进行分组，如图 4.6 所示。除此之外，还需创建一个称作 Agent 的立方体对象，并赋予障碍物躲避脚本。随后，可构建一个地面平面，这将有助于搜寻目标位置。

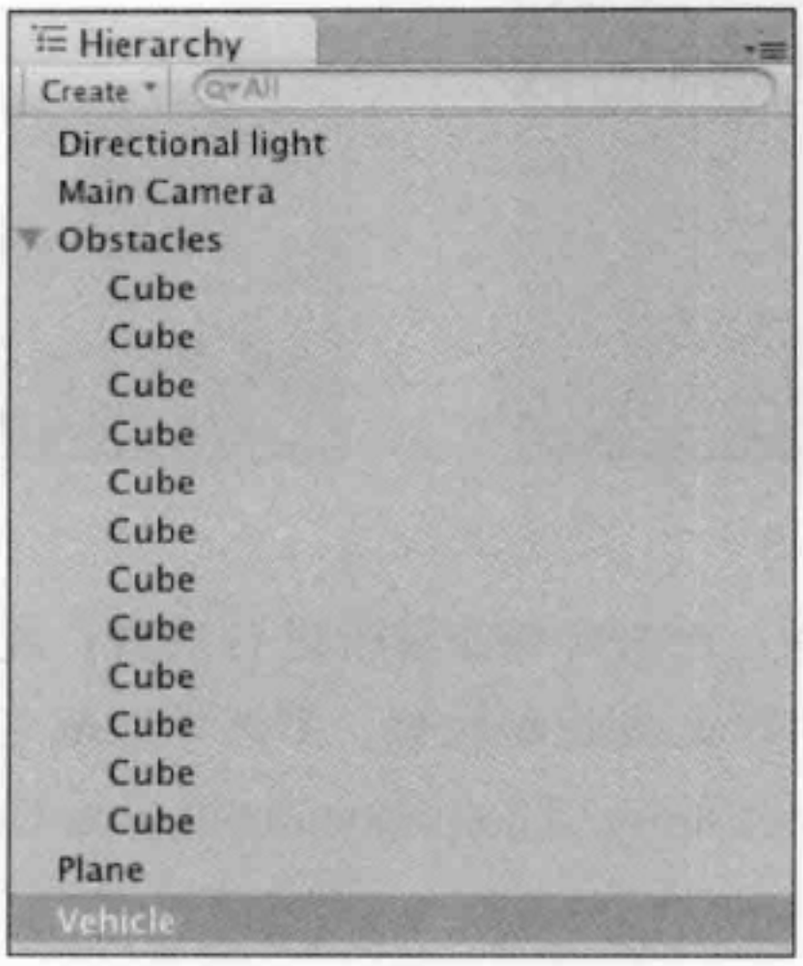

图 4.6

需要注意的是，上述 Agent 对象并非是路径搜索器。同样，如果设置过多的墙体，这将大大提升 Agent 对象搜索目标的难度。对此，可先期尝试少量墙体，并查看 Agent 对象的运行方式。

4.1.4 添加定制层

下面将向当前对象添加定制层。当添加某一新层时，可选择 Edit | Project Settings | Tags 选项，并将 User Layer 8 设置为 Obstacles，如图 4.7 所示。相应地，还需返回至立方体实体处，并将其层属性设置为 Obstacles，如图 4.8 所示。

上述新层将添加至 Unity 中，当执行光线投射操作检测障碍物时，仅需通过该特定层对此类实体进行检测。通过这一方式，可忽略与光线碰撞的非障碍物对象，例如灌木丛或菜地。

对于大型项目，游戏对象可能已经包含了赋予其中的某一层，因而无须将对象层调整为 Obstacles。相反，可针对立方体实体采用层位图生成一个列表，以供障碍物检测时使用。稍后将对位图加以讨论。

图 4.7

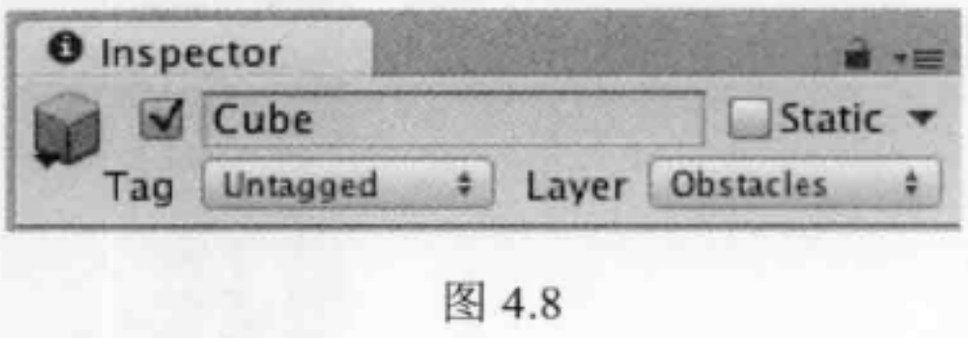

图 4.8

技巧：层常用于相机对象中，进而对部分场景进行渲染；或者通过光源照亮部分场景。除此之外，层还可用于光线投射机制，并选择性地忽略某些碰撞行为。关于层，读者可访问http://docs.unity3d.com/Documentation/Components/Layers.html以获取更多内容。

4.1.5　实现躲避逻辑

下面将编写脚本文件，以帮助立方体实体躲避墙体，如图 4.9 所示。

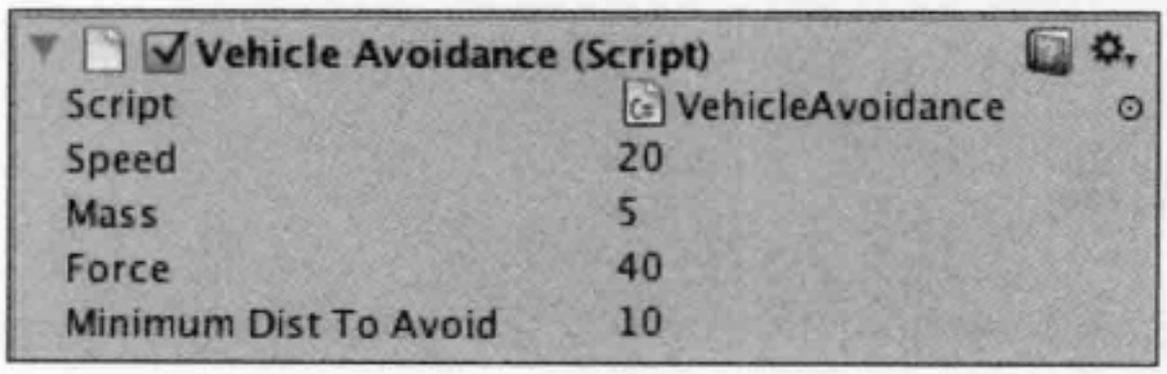

图 4.9

通常情况下，可采用默认属性初始化实体脚本，并在 OnGUI 方法中绘制 GUI 文本。下列内容显示了 VehicleAvoidance.cs 文件中的代码：

```
using UnityEngine;
using System.Collections;

public class VehicleAvoidance : MonoBehaviour {
  public float speed = 20.0f;
```

```
public float mass = 5.0f;
public float force = 50.0f;
public float minimumDistToAvoid = 20.0f;

//Actual speed of the vehicle
private float curSpeed;
private Vector3 targetPoint;

//Use this for initialization
void Start () {
  mass = 5.0f;
  targetPoint = Vector3.zero;
}

void OnGUI() {
  GUILayout.Label("Click anywhere to move the vehicle.");
}
```

在 Update 方法中，将根据 AvoidObstacles 方法返回的方向向量更新主体实体的位置和旋转，如下所示：

```
//Update is called once per frame
void Update () {
  //Vehicle move by mouse click
  RaycastHit hit;
var ray = Camera.main.ScreenPointToRay
    (Input.mousePosition);

if (Input.GetMouseButtonDown(0) &&
  Physics.Raycast(ray, out hit, 100.0f)) {
  targetPoint = hit.point;
}

//Directional vector to the target position
Vector3 dir = (targetPoint - transform.position);
dir.Normalize();
```

```
    //Apply obstacle avoidance
    AvoidObstacles(ref dir);

    //...

}
```

在 Update 方法中，首先需要获取鼠标的点击位置，进而移动 AI 实体。对此，可在相机视见方向上投射一条光线。随后，可使用光线与地面间的碰撞点作为目标位置。当获取目标位置向量后，即可计算方向向量，即从目标位置向量中减去当前位置向量。随后，可调用 AvoidObstacles 方法，并传入该方向向量，如下所示：

```
//Calculate the new directional vector to avoid
  //the obstacle
public void AvoidObstacles(ref Vector3 dir) {
  RaycastHit hit;

  //Only detect layer 8 (Obstacles)
  int layerMask = 1<<8;

  //Check that the vehicle hit with the obstacles within
    //it's minimum distance to avoid
  if (Physics.Raycast(transform.position,
    transform.forward, out hit,

      minimumDistToAvoid, layerMask)) {
    //Get the normal of the hit point to calculate the
      //new direction
      Vector3 hitNormal = hit.normal;
      hitNormal.y = 0.0f; //Don't want to move in Y-Space

      //Get the new directional vector by adding force to
      //vehicle's current forward vector
    dir = transform.forward + hitNormal * force;
  }
 }
}
```

AvoidObstacles 方法同样十分简单，此处唯一需要注意的技巧是，光线投射有选择性地与 Obstacles 层相交，如前所述，该层定义于 Unity TagManager 中的 User Layer 8 处。Raycast 方法接收一个层遮罩参数，进而在光线投射过程中确定需要忽略或考察的对应层。目前，当查看所定义的全部层时，可看到共计 32 层。因此，Unity 使用 32 位整数表示层遮罩参数。例如，下列 32 位数据表示数字 0：

```
0000 0000 0000 0000 0000 0000 0000 0000
```

默认情况下，Unity 使用前 8 个层作为内建层。因此，若未采用层遮罩参数自行光线投射操作时，该操作将涉及全部 8 个层，对应的位掩码如下所示：

```
0000 0000 0000 0000 0000 0000 1111 1111
```

此处的 Obstacles 层设置于 layer 8 处（第 9 个索引），且仅需要向该层投射光线，因而可按照下列方式设置位掩码：

```
0000 0000 0000 0000 0000 0001 0000 0000
```

设置位掩码最为简单的方式是使用移位操作符。此处仅需要在第 9 个索引处设置“on”或 1，即左移 8 位。对此，可采用左移操作符向左移动该位 8 位，如下所示：

```
int layerMask = 1<<8;
```

当采用多个层遮罩时，例如 layer 8 和 layer 9，一种简便的方法是使用 OR 位操作符，如下所示：

```
int layerMask = (1<<8) | (19);
```

提示： 关于层遮罩的更多讨论，读者可访问 Unity 3D 的在线资源。另外，读者还可访问 http://answers.unity3d.com/questions/8715/howdo-i-use-layermasks.html，并查看与 uci 相关的问题和回复。

当层遮罩设置完毕后，可调用根据实体的当前位置并在前向方向上调用 Physics.Raycast 方法。对于光线长度，此处使用了变量 minimumDistToAvoid，也就是说，仅躲避位于该距离内与光线相交的障碍物。

随后，可使用碰撞光线的规范化向量，乘以作用力向量，将其加至实体的当前方向上，进而得到最终的方向向量，并由当前方法予以返回，如图 4.10 所示。

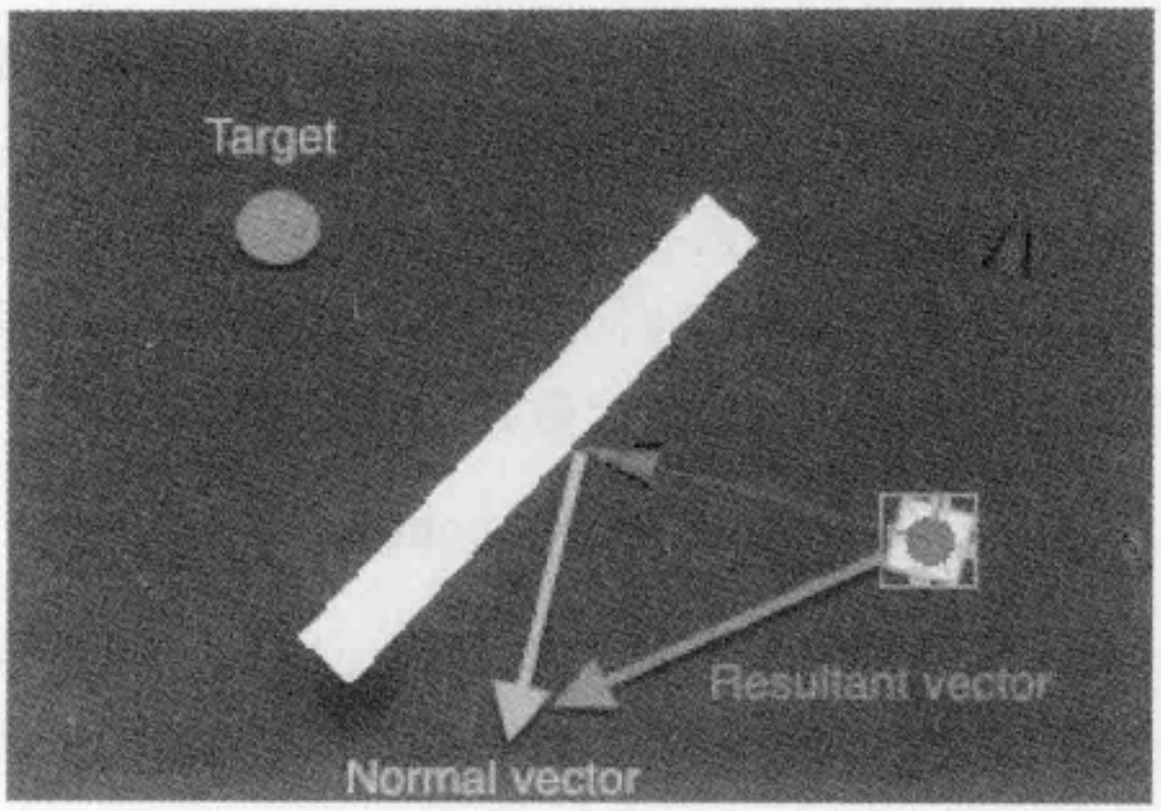

图 4.10

随后，Update 方法在躲避障碍物后使用上述新向量旋转 AI 实体，并根据速度值更新位置，如下所示：

```
void Update () {

  //...

  //Don't move the vehicle when the target point
    //is reached
  if (Vector3.Distance(targetPoint,
    transform.position) < 3.0f) return;

    //Assign the speed with delta time
    curSpeed = speed * Time.deltaTime;

    //Rotate the vehicle to its target
      //directional vector
var rot = Quaternion.LookRotation(dir);
transform.rotation = Quaternion.Slerp
    (transform.rotation, rot, 5.0f *
  Time.deltaTime);

  //Move the vehicle towards
```

```
        transform.position += transform.forward *
          curSpeed;
    }
```

4.2 A*寻路算法

下面将采用 C#语言实现 Unity 环境中的 A*算法。考虑到简单性和高效性，A*寻路算法广泛地应用于游戏和交互式应用程序中。当然，除此之外还存在其他寻路算法，例如 Dijkstra 算法。第 1 章曾简要介绍了该 A*算法，下面将从实现角度再次对该算法进行回顾。

4.2.1 算法回顾

关于 A*算法，首先需要通过某种数据结构表示地图，对此存在多种结构可完成这一项任务。此处使用了 2D 网格数组，稍后将实现 GridManager 类，并处理此类地图信息。GridManager 类维护了一个 Node 对象列表，基本上表示为 2D 网格单元。因此，需要实现 Node 类，进而处理节点类型（无论是可穿越节点亦或是障碍物节点）、路径估值以及到达目标 Node 的估值等。

此处采用两个变量存储处理完毕的节点和行将处理的节点，并分别将其称作闭合表和开放表，PriorityQueue 类中将实现该表类型。最后，A*算法将在 AStar 类中实现，对应步骤如下：

（1）将起始节点置于开放列表中。

（2）若开放列表中包含节点，则执行下列处理步骤。

- 从开放列表中拾取首个节点，并使其保持为当前节点（假设已经存储了开放列表，且首节点包含最小估值，代码结尾处将对此有所提及）。
- 获取当前节点的邻接节点（非障碍物类型），例如无法穿越的墙体或峡谷。
- 针对各个邻接节点，检测该邻接节点是否已位于闭合表中。若否，则使用下列公式计算整体估值（F）：

$$F = G + H$$

在上述公式中。G 表示为上一节点至当前节点间的整体估值；H 表示为当前节点至最终目标节点间的整体估值。

- 在邻接节点对象中存储上述估值数据。另外，还需要将当前节点存储为父节点。稍后，将采用该父节点数据回溯实际路径。
- 将上述邻接节点置于开放列表中。随后，按照至目标节点的整体估值对开放列表进行升序排序。
- 如果邻接节点处理完毕，可将当前节点置于闭合列表中，并将其从开放列表中移除。
- 返回至步骤（2）。

一旦完成了上述处理过程，当前节点应位于目标节点位置处，但仅存在一条无障碍路径可从起始节点到达目标节点。如果当前节点未处于目标节点，则不存在一条至目标节点的有效路径。对于有效路径，全部工作即是从当前节点的父节点处执行回溯操作，直至再次到达起始节点。这将生成一个寻路过程中所选取的、全部节点的路径列表，其顺序为目标节点至起始节点。随后，鉴于需要获得起始节点至目标节点间的路径，因而逆置该路径列表即可。

上述步骤展示了算法的整体描述，下面将讨论 Unity 中基于 C#语言的实现过程。

4.2.2　算法实现

本节将实现之前讨论的某些基本类，例如 Node 类、GridManager 类以及 PriorityQueue 类。随后，将在主类 AStar 中对其予以应用。

1. 实现 Node 类

Node 类用于处理 2D 网格中的各个单元对象，并以此表示 Node.cs 文件中的地图，如下所示：

```
using UnityEngine;
using System.Collections;
using System;

public class Node : IComparable {
  public float nodeTotalCost;
  public float estimatedCost;
  public bool bObstacle;
  public Node parent;
```

```
public Vector3 position;

public Node() {
  this.estimatedCost = 0.0f;
  this.nodeTotalCost = 1.0f;
  this.bObstacle = false;
  this.parent = null;
}

public Node(Vector3 pos) {
  this.estimatedCost = 0.0f;
  this.nodeTotalCost = 1.0f;
  this.bObstacle = false;
  this.parent = null;
  this.position = pos;
}

public void MarkAsObstacle() {
  this.bObstacle = true;
}
```

Node 类中包含了多个属性，例如估值数据（G 和 H）、障碍物标记、位置以及父节点。其中，nodeTotalCost 表示为 G，即起始节点和当前节点之间的移动估值；estimatedCost 表示为 H，即当前节点至目标节点之间的整体估算值。另外，该类中还定义了两个简单的构造方法以及一个封装方法，并可将节点设置为障碍物对象。随后，可实现 CompareTo 方法，如下所示：

```
public int CompareTo(object obj) {
  Node node = (Node)obj;
  //Negative value means object comes before this in the sort
    //order.
  if (this.estimatedCost < node.estimatedCost)
    return -1;
  //Positive value means object comes after this in the sort
    //order.
  if (this.estimatedCost > node.estimatedCost) return 1;
```

```
        return 0;
        }
    }
```

该方法较为重要。由于需要覆写 CompareTo 方法，因而当前 Node 类继承自 IComparable 类。针对前述讨论，需要注意应根据整体估算值对节点数组列表进行排序。相应地，ArrayList 类型包含了名为 Sort 的方法，该方法查找列表中 Node 对象中实现的 CompareTo 方法。因此，此处根据 estimatedCost 值对节点对象进行排序。

提示： IComparable.CompareTo 方法体现了.NET 框架特征，读者可访问 http://msdn.microsoft.com/en-us/library/system.icomparable.compareto.aspx 以获取更多信息。

2. 构建优先队列

PriorityQueue 类可方便地处理节点的 ArrayList，对应代码位于 PriorityQueue.cs 类中，如下所示：

```
using UnityEngine;
using System.Collections;

public class PriorityQueue {

  private ArrayList nodes = new ArrayList();

  public int Length {
    get { return this.nodes.Count; }
  }

public bool Contains(object node) {
  return this.nodes.Contains(node);
}

public Node First() {
  if (this.nodes.Count > 0) {
    return (Node)this.nodes[0];
  }
  return null;
```

```
}

public void Push(Node node) {
  this.nodes.Add(node);
  this.nodes.Sort();
}

public void Remove(Node node) {
  this.nodes.Remove(node);
  //Ensure the list is sorted
    this.nodes.Sort();
  }
}
```

上述代码内容并不复杂，需要注意的是，当向节点的 ArrayList 中添加节点或从中移除节点后，需要调用 Sort 方法。这将调用 Node 对象的 CompareTo 方法，并通过 estimatedCost 值对节点进行排序。

3. 创建网格管理器

GridManager 类处理网格的全部属性，并以此体现地图特征。由于仅需要单一对象表示地图，因而此处持有 GridManager 类的单体实例。对应代码位于 GridManager.cs 文件中，如下所示：

```
using UnityEngine;
using System.Collections;

public class GridManager : MonoBehaviour {

private static GridManager s_Instance = null;

public static GridManager instance {
  get {
    if (s_Instance == null) {
      s_Instance = FindObjectOfType(typeof(GridManager))
          as GridManager;
      if (s_Instance == null)
```

```
            Debug.Log("Could not locate a GridManager " +
                "object. \n You have to have exactly " +
                "one GridManager in the scene.");
        }
        return s_Instance;
      }
    }
```

在当前场景中，可查找 GridManager 对象，对应结果保存于 s_Instance 静态变量中，如下所示：

```
public int numOfRows;
public int numOfColumns;
public float gridCellSize;
public bool showGrid = true;
public bool showObstacleBlocks = true;

private Vector3 origin = new Vector3();
private GameObject[] obstacleList;
public Node[,] nodes { get; set; }
public Vector3 Origin {
  get { return origin; }
}
```

随后将声明全部变量，并体现地图的全部内容，例如行和列的数量、各个网格单元的尺寸以及某些 Boolean 变量（对网格和障碍物实现可视化操作，存储网格中的全部节点），对应代码如下所示：

```
void Awake() {
  obstacleList = GameObject.FindGameObjectsWithTag("Obstacle");
  CalculateObstacles();
}
//Find all the obstacles on the map
void CalculateObstacles() {
  nodes = new Node[numOfColumns, numOfRows];
  int index = 0;
```

```
for (int i = 0; i < numOfColumns; i++) {
  for (int j = 0; j < numOfRows; j++) {
    Vector3 cellPos = GetGridCellCenter(index);
    Node node = new Node(cellPos);
    nodes[i, j] = node;
    index++;
  }
}
if (obstacleList != null && obstacleList.Length > 0) {
  //For each obstacle found on the map, record it in our list
    foreach (GameObject data in obstacleList) {
      int indexCell = GetGridIndex(data.transform.position);
      int col = GetColumn(indexCell);
      int row = GetRow(indexCell);
      nodes[row, col].MarkAsObstacle();
    }
  }
}
```

此处利用 Obstacle 标签查找全部游戏对象，并将其置于 obstacleList 属性中。随后在 CalculateObstacles 方法中设置节点的 2D 数组。首先，需要利用默认属性生成标准的节点对象，并于随后检测 obstacleList，将其位置转换为行、列数据，进而将对应索引处的节点更新为障碍物。

GridManager 类定义了多个辅助方法，遍历网格并获取网格单元数据。其实现过程较为简单，因而这里并不打算对其进行深入讨论。

GetGridCellCenter 方法从网格单元索引处返回世界坐标系中该单元的位置，如下所示：

```
public Vector3 GetGridCellCenter(int index) {
  Vector3 cellPosition = GetGridCellPosition(index);
  cellPosition.x += (gridCellSize / 2.0f);
  cellPosition.z += (gridCellSize / 2.0f);
  return cellPosition;
}
```

```
public Vector3 GetGridCellPosition(int index) {
  int row = GetRow(index);
  int col = GetColumn(index);
  float xPosInGrid = col * gridCellSize;
  float zPosInGrid = row * gridCellSize;
  return Origin + new Vector3(xPosInGrid, 0.0f, zPosInGrid);
}
```

GetGridIndex 方法根据既定位置返回网格中的单元索引，如下所示：

```
public int GetGridIndex(Vector3 pos) {
  if (!IsInBounds(pos)) {
    return -1;
  }
  pos -= Origin;
  int col = (int)(pos.x / gridCellSize);
  int row = (int)(pos.z / gridCellSize);
  return (row * numOfColumns + col);
}

public bool IsInBounds(Vector3 pos) {
  float width = numOfColumns * gridCellSize;
  float height = numOfRows* gridCellSize;
  return (pos.x >= Origin.x && pos.x <= Origin.x + width &&
      pos.x <= Origin.z + height && pos.z >= Origin.z);
}
```

GetRow 和 GetColumn 方法根据既定索引返回网格单元的行、列数据，如下所示：

```
public int GetRow(int index) {
  int row = index / numOfColumns;
  return row;
}

public int GetColumn(int index) {
  int col = index % numOfColumns;
  return col;
}
```

GetNeighbours 则是另一个较为重要的方法，在 AStar 类中，该方法用于获得特定解的邻接节点，如下所示：

```
public void GetNeighbours(Node node, ArrayList neighbors) {
  Vector3 neighborPos = node.position;
  int neighborIndex = GetGridIndex(neighborPos);

  int row = GetRow(neighborIndex);

  int column = GetColumn(neighborIndex);

  //Bottom
  int leftNodeRow = row - 1;
  int leftNodeColumn = column;
  AssignNeighbour(leftNodeRow, leftNodeColumn, neighbors);

  //Top
  leftNodeRow = row + 1;
  leftNodeColumn = column;
  AssignNeighbour(leftNodeRow, leftNodeColumn, neighbors);

  //Right
  leftNodeRow = row;
  leftNodeColumn = column + 1;
  AssignNeighbour(leftNodeRow, leftNodeColumn, neighbors);

  //Left
  leftNodeRow = row;
  leftNodeColumn = column - 1;
  AssignNeighbour(leftNodeRow, leftNodeColumn, neighbors);
}

void AssignNeighbour(int row, int column, ArrayList neighbors) {
  if (row != -1 && column != -1 &&
      row < numOfRows && column < numOfColumns) {
    Node nodeToAdd = nodes[row, column];
```

```
        if (!nodeToAdd.bObstacle) {
          neighbors.Add(nodeToAdd);
        }
      }
    }
```

首先，该方法获取当前节点上、下、左、右全部方向上的邻接节点。随后，在 AssignNeighbour 方法内将对节点进行检测并查看是否为障碍物对象。若否，可将邻接节点置于引用数组列表 neighbors 中。出于调试目的，接下来的方法对网格和障碍物执行可视化操作，如下所示：

```
void OnDrawGizmos() {
  if (showGrid) {
    DebugDrawGrid(transform.position, numOfRows, numOfColumns,
        gridCellSize, Color.blue);
  }

    Gizmos.DrawSphere(transform.position, 0.5f);
    if (showObstacleBlocks) {
      Vector3 cellSize = new Vector3(gridCellSize, 1.0f,
        gridCellSize);
      if (obstacleList != null && obstacleList.Length > 0) {
        foreach (GameObject data in obstacleList) {
          Gizmos.DrawCube(GetGridCellCenter(
              GetGridIndex(data.transform.position)), cellSize);
        }
      }
  }
}

public void DebugDrawGrid(Vector3 origin, int numRows, int
  numCols,float cellSize, Color color) {
  float width = (numCols * cellSize);
  float height = (numRows * cellSize);

  //Draw the horizontal grid lines
```

```
for (int i = 0; i < numRows + 1; i++) {
  Vector3 startPos = origin + i * cellSize * new Vector3(0.0f,
    0.0f, 1.0f);
  Vector3 endPos = startPos + width * new Vector3(1.0f, 0.0f,
    0.0f);
  Debug.DrawLine(startPos, endPos, color);
}

//Draw the vertical grid lines
for (int i = 0; i < numCols + 1; i++) {
    Vector3 startPos = origin + i * cellSize * new Vector3(1.0f,
      0.0f, 0.0f);
    Vector3 endPos = startPos + height * new Vector3(0.0f, 0.0f,
      1.0f);
    Debug.DrawLine(startPos, endPos, color);
  }
 }
}
```

线框用于绘制可视化的调试结果，并在编辑器场景视图中设置辅助内容。OnDrawGizmos 方法在每帧中通过引擎被调用，因此，如果调试标记 showGrid 和 showObstacleBlocks 被选中，则采用直线绘制网格，并通过立方体绘制障碍物立方体对象。相应地，鉴于 DebugDrawGrid 方法较为简单，因而此处不予赘述。

提示：关于线框的更多信息，读者可阅读 Unity 的参考文档，对应网址为 http://docs.unity3d.com/Documentation/ScriptReference/Gizmos.html。

4. A*算法的实现细节

AStar 定义为主类，并利用了之前定义的多个类。在 AStar.cs 文件中，openList 和 closedList 声明为 PriorityQueue 类型，如下所示：

```
using UnityEngine;
using System.Collections;

public class AStar {
  public static PriorityQueue closedList, openList;
```

随后将实现 HeuristicEstimateCost 方法，并计算两个节点间的估值。对应计算过程较为简单，仅需获取两个节点间的方向向量，即在位置向量间执行减法运算。结果向量值表示为当前节点至目标节点间的直接距离，如下所示：

```
private static float HeuristicEstimateCost(Node curNode,
    Node goalNode) {
  Vector3 vecCost = curNode.position - goalNode.position;
  return vecCost.magnitude;
}
```

下面是 FindPath 主方法，如下所示：

```
public static ArrayList FindPath(Node start, Node goal) {
  openList = new PriorityQueue();
  openList.Push(start);
  start.nodeTotalCost = 0.0f;
  start.estimatedCost = HeuristicEstimateCost(start, goal);

  closedList = new PriorityQueue();
  Node node = null;
```

下面对开放列表和闭合表执行初始化操作。首先是起始节点，可将其置于开放列表中，并于随后对开放列表加以处理，如下所示：

```
while (openList.Length != 0) {
  node = openList.First();
//Check if the current node is the goal node

  if (node.position == goal.position) {
    return CalculatePath(node);
  }

  //Create an ArrayList to store the neighboring nodes
  ArrayList neighbours = new ArrayList();

  GridManager.instance.GetNeighbours(node, neighbours);
```

```
    for (int i = 0; i < neighbours.Count; i++) {
      Node neighbourNode = (Node)neighbours[i];

      if (!closedList.Contains(neighbourNode)) {
        float cost = HeuristicEstimateCost(node,
            neighbourNode);

        float totalCost = node.nodeTotalCost + cost;
        float neighbourNodeEstCost = HeuristicEstimateCost(
            neighbourNode, goal);

        neighbourNode.nodeTotalCost = totalCost;
        neighbourNode.parent = node;
        neighbourNode.estimatedCost = totalCost +
            neighbourNodeEstCost;

        if (!openList.Contains(neighbourNode)) {
          openList.Push(neighbourNode);
        }
      }
    }
    //Push the current node to the closed list
    closedList.Push(node);
    //and remove it from openList
    openList.Remove(node);
  }

  if (node.position != goal.position) {
    Debug.LogError("Goal Not Found");
    return null;
  }
  return CalculatePath(node);
}
```

与之前算法相比，代码实现基本类似，并执行了下列操作步骤：

（1）获取 openList 中的第一个节点。回忆一下，当每次添加新节点时，节点的

openList 通常呈有序状态。因此，相对于目标节点，首个节点通常为包含最小估值的节点。

（2）检测当前节点是否已处于目标节点位置。若是，则退出 while 循环，并构建路径数组。

（3）创建数组列表，存储经处理的、当前节点的邻接节点。利用 GetNeighbours 方法从网格中获取邻接节点。

（4）针对邻接节点数组中的各个节点，检测是否已位于 closedList 中。若否，则计算估值，利用新估值以及父节点数据更新该节点的属性，并将其置于 openList 中。

（5）将当前节点置于 closedList 中，并将其从 openList 中移除。返回至步骤（1）。

如果 openList 为空，且存在一条有效的路径，则当前节点位于目标节点处。随后，可利用当前节点参数调用 CalculatePath 方法，如下所示：

```
    private static ArrayList CalculatePath(Node node) {
      ArrayList list = new ArrayList();
      while (node != null) {
        list.Add(node);
        node = node.parent;
      }
      list.Reverse();
      return list;
    }
  }
```

CalculatePath 方法跟踪各个节点的父节点对象，并创建数组列表，进而生成目标节点至起始节点间的节点数组列表。鉴于路径通常表示为起始节点至目标节点，因而此处需要调用 Reverse 方法。

至此，AStar 类的定义暂告一段落。下面将编写脚本对其进行测试，并于随后构建相应的场景以实现 AStar 类的应用。

5. 实现 TestCode 类

该类使用 AStar 类获取起始节点至目标节点之间的路径，对应内容位于 TestCode.cs 文件中，如下所示：

```
using UnityEngine;
using System.Collections;

public class TestCode : MonoBehaviour {
  private Transform startPos, endPos;
  public Node startNode { get; set; }
  public Node goalNode { get; set; }

  public ArrayList pathArray;

  GameObject objStartCube, objEndCube;
  private float elapsedTime = 0.0f;
  //Interval time between pathfinding
  public float intervalTime = 1.0f;
```

代码首先设置所需引用的变量。其中，pathArray 存储返回自 AStar FindPath 方法的节点数组，如下所示：

```
void Start () {
  objStartCube = GameObject.FindGameObjectWithTag("Start");
  objEndCube = GameObject.FindGameObjectWithTag("End");

  pathArray = new ArrayList();
  FindPath();
}

void Update () {
  elapsedTime += Time.deltaTime;
  if (elapsedTime >= intervalTime) {
    elapsedTime = 0.0f;
    FindPath();
  }
}
```

在 Start 方法中，将利用 Start 和 End 标签查找对象，并初始化 pathArray。在各间隔（通过 intervalTime 属性进行设置）处，将尝试获取新路径，以防起始和终止节点的位置发生变化。随后将调用 FindPath 方法，如下所示：

```
void FindPath() {
  startPos = objStartCube.transform;
  endPos = objEndCube.transform;

  startNode = new Node(GridManager.instance.GetGridCellCenter(
     GridManager.instance.GetGridIndex(startPos.position)));

  goalNode = new Node(GridManager.instance.GetGridCellCenter(
     GridManager.instance.GetGridIndex(endPos.position)));

  pathArray = AStar.FindPath(startNode, goalNode);
}
```

考虑到寻路算法实现于 AStar 类中，因而获取一条路径将变得十分简单。首先，可获取起始和终止游戏对象的位置。随后，可通过 GridManager 和 GetGridIndex 辅助方法创建新 Node 对象，并计算网格内相应的行、列索引位置。一旦上述工作执行完毕，即可通过起始节点和目标节点调用 AStar.FindPath 方法，并将返回的数组列表存储于 pathArray 局部属性中。随后，将 OnDrawGizmos 实现方法，并对结果路径进行绘制，如下所示：

```
void OnDrawGizmos() {
  if (pathArray == null)
    return;

  if (pathArray.Count > 0) {
    int index = 1;
    foreach (Node node in pathArray) {
      if (index < pathArray.Count) {
        Node nextNode = (Node)pathArray[index];
        Debug.DrawLine(node.position, nextNode.position,
          Color.green);
        index++;
      }
    }
  }
}
```

代码遍历 pathArray，并通过 Debug.DrawLine 方法绘制源自 pathArray 的、用以连接节点的直线。据此，当运行并测试场景时，可看到起始节点和终止节点之间的、连接多个节点的一条直线，最终形成了一条有效路径。

6. 构建示例场景

下面将构建如图 4.11 所示的场景。

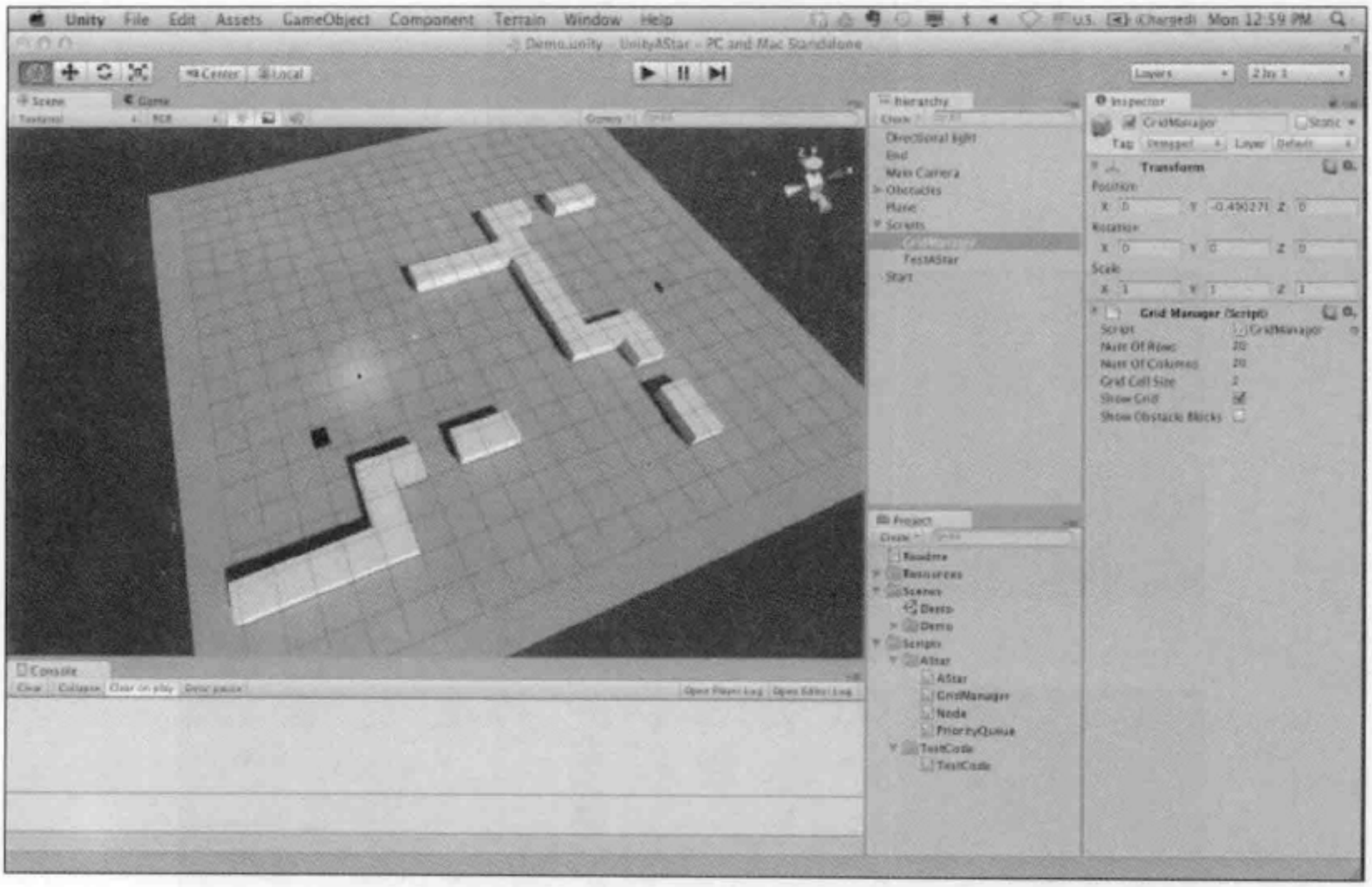

图 4.11

相应地，这里将构建有向光源、起始和目标游戏对象、障碍物对象，用于表示地面的平面实体，以及放置 GridManager 和 TestAStar 脚本的两个空游戏对象。对应层次结构如图 4.12 所示。

另外，还需创建立方体实体，并将其标记为 Obstacle。当运行寻路算法时，将通过此类标记查找对象，如图 4.13 所示。

创建立方体实体并将其标记为 Start，如图 4.14 所示。

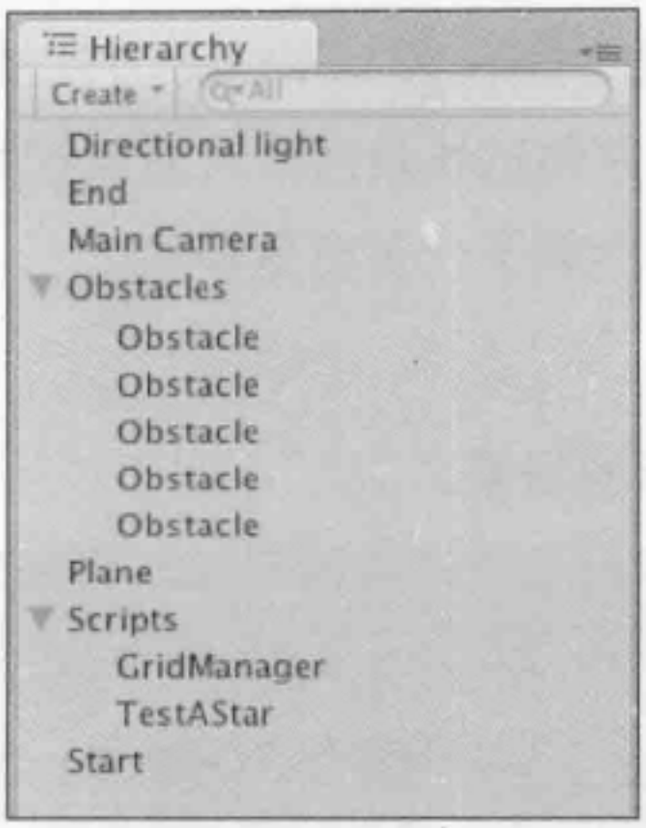

图 4.12

图 4.13

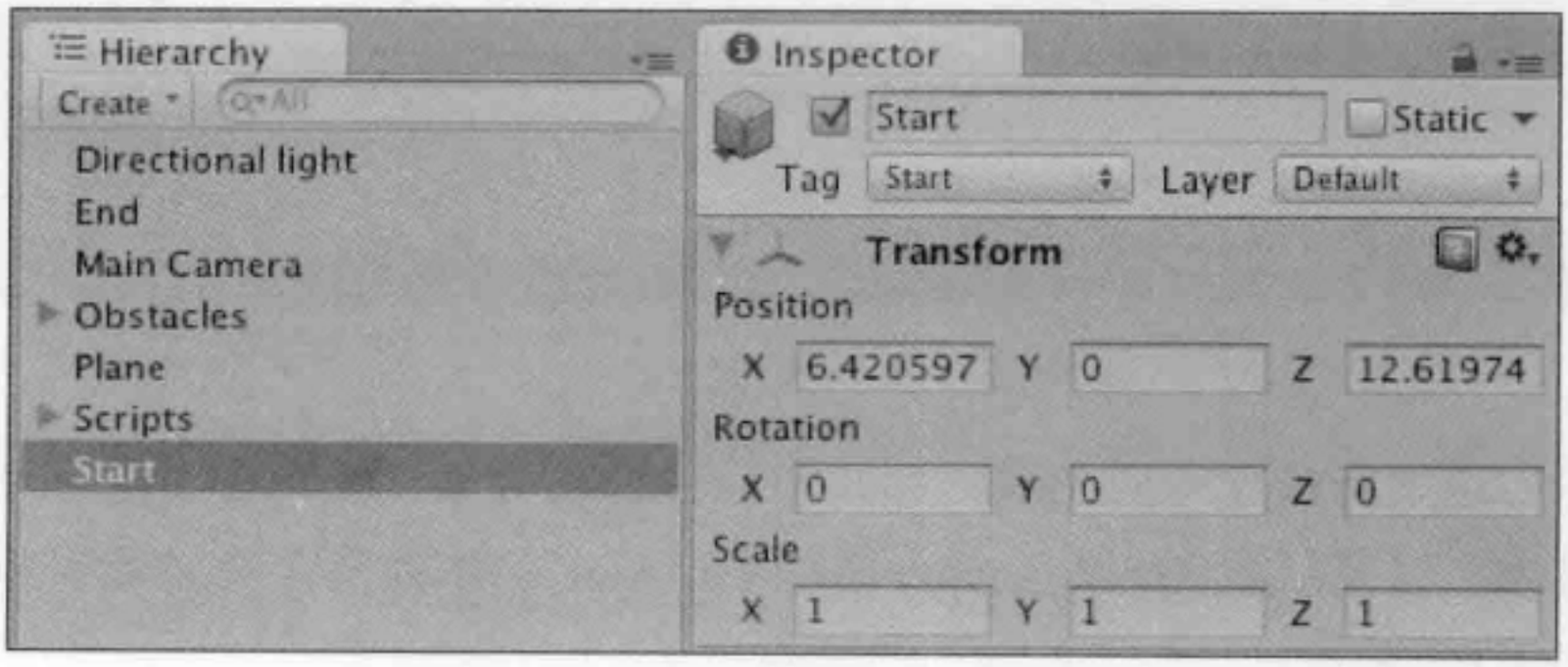

图 4.14

随后，创建另一个立方体实体，并将其标记为 End，如图 4.15 所示。

当前，还需要创建空游戏对象，并绑定 GridManager 脚本。由于需要通过相关名称在脚本中查找 GridManager 对象，因而此处应将名称项设置为 GridManager，如图 4.16

所示。除此之外，这里还可设置网格的行、列数量，以及各个网格单元的尺寸。

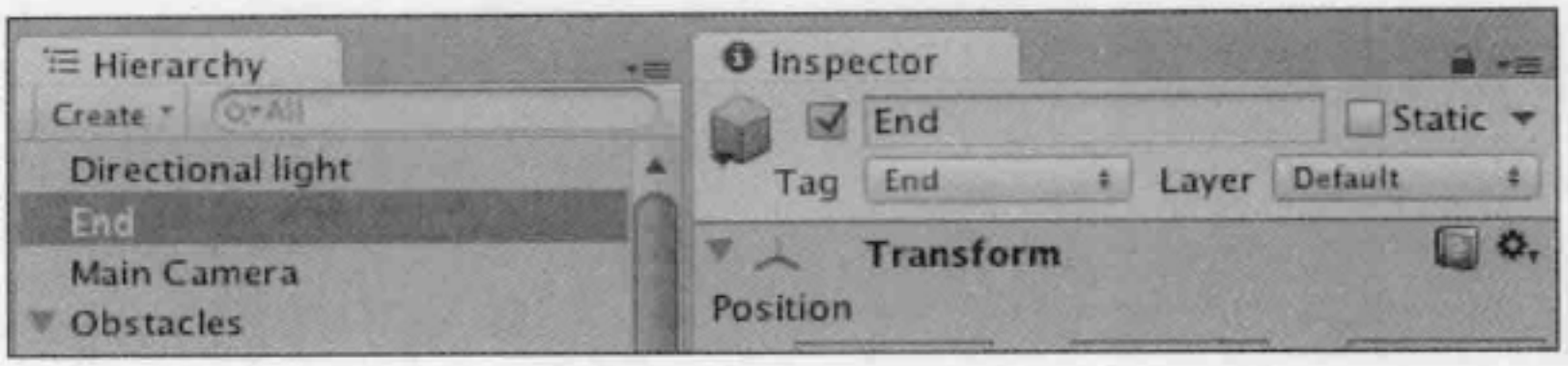

图 4.15

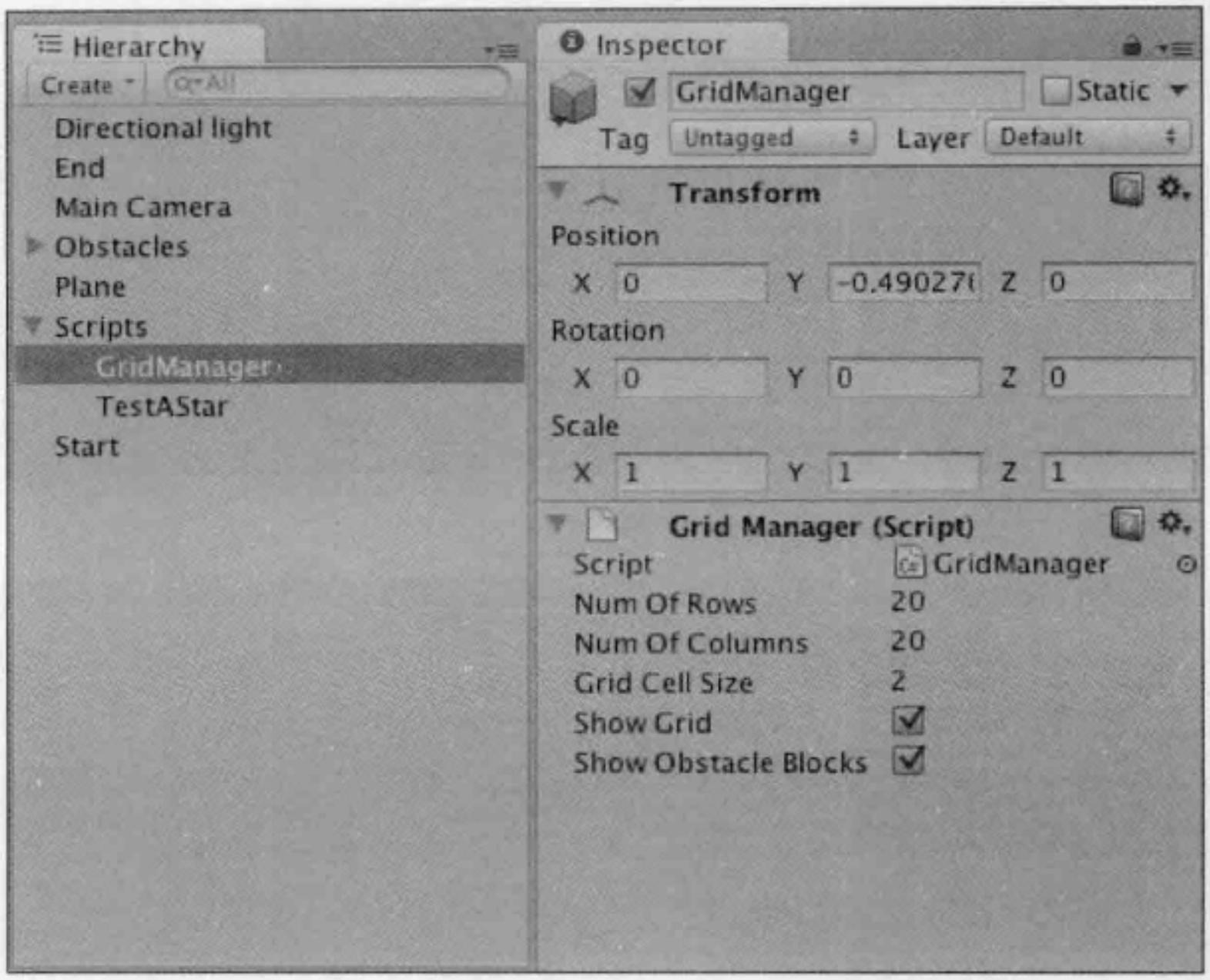

图 4.16

7. 测试全部组件

最后，单击播放按钮，并实际查看 A*寻路算法的运行效果。默认条件下，当场景被播放时，Unity 将切换至 Game 视图中。由于寻路可视化代码针对调试模式编写，也就是说，在编辑器视图中进行绘制，因而此处需要切换回 Scene 视图，或启用 Gizmos 以查看结果路径，如图 4.17 所示。

另外，读者还可通过编辑器的移动线框（位于 Scene 视图中，而非 Game 视图）尝试移动场景中的起始节点和目标节点，如图 4.18 所示。

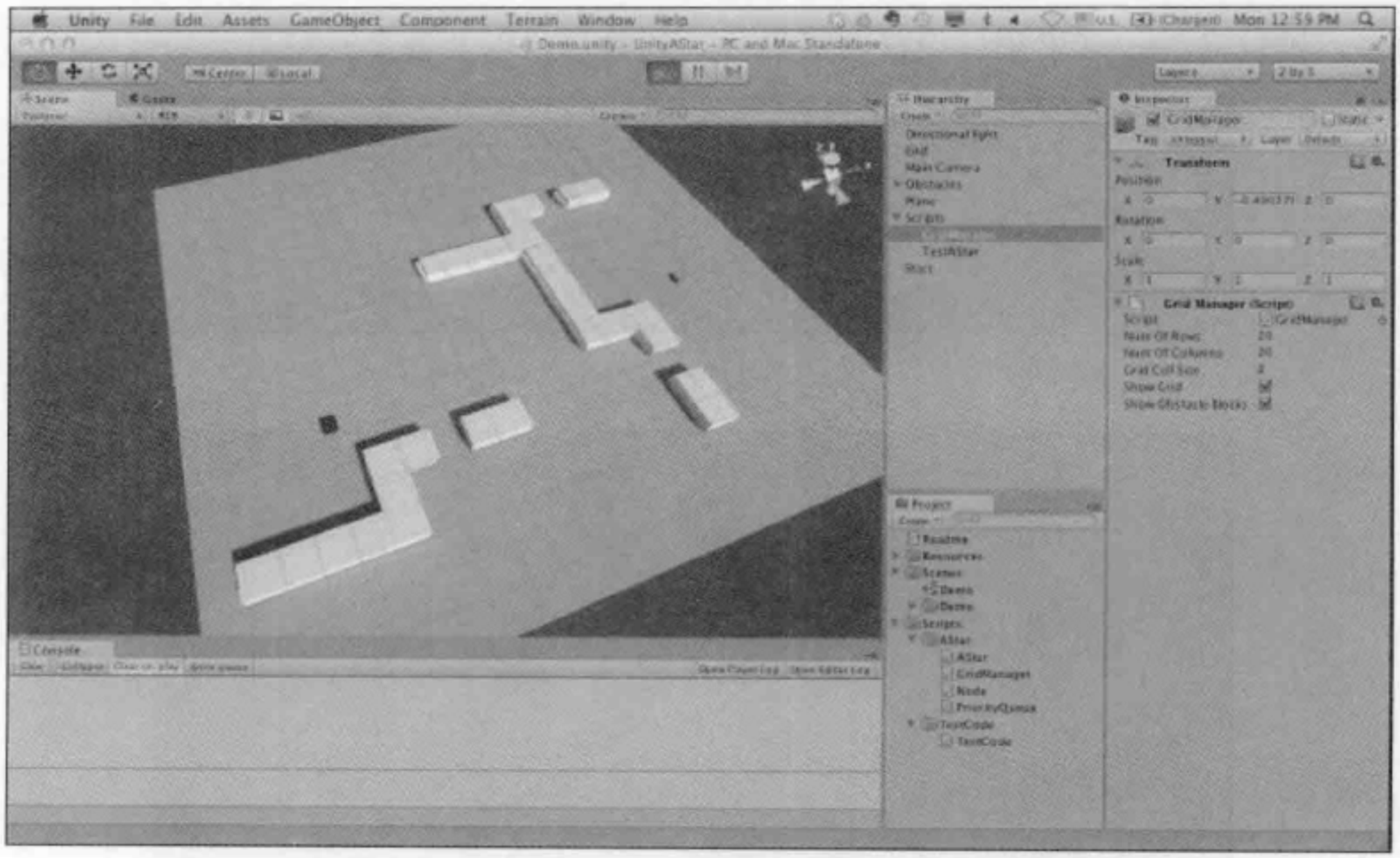

图 4.17

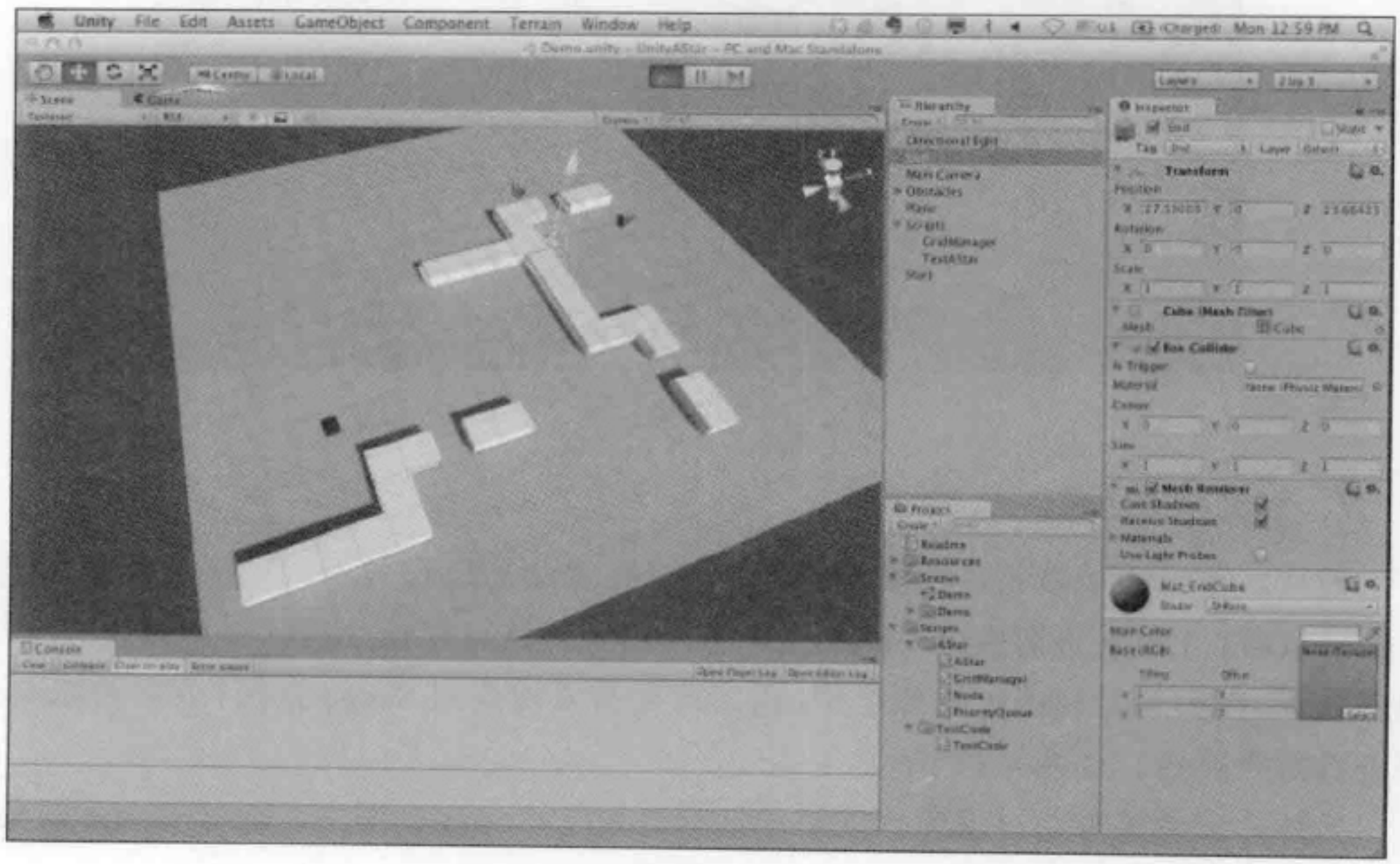

图 4.18

通过观察可知，如果起始节点和目标节点之间存在一条有效路径，则路径将以实时动态方式被更新。如果路径无效，控制台窗口中将显示一条错误消息。

4.3 导 航 网 格

本节讨论 Unity 内建的导航网格生成器，进而简化 AI 主体对象的寻路方式。在 Unity 5 版本中，NavMesh 可供全部用户使用。之前，NavMesh 仅是 Unity Pro 中的高级特性，现在则演变为 Unity 个人版中的组件。第 2 章简要讨论了 Unity 中的 NavMesh 组件，其中涉及了有限状态机测试过程中 NavMesh 主体对象的运动行为。本节将深入讨论该系统的全部内容。AI 寻路机制需要通过特定格式对场景进行描述，前述内容已经介绍了 2D 地图上 A*寻路算法中的 2D 网格（数组）。其中，AI 主体对象需要了解障碍物所处的位置，特别是静态障碍物。动态运动对象间的碰撞躲避问题则是另一个主题，即物体的转向行为。Unity 包含了内建的导航特性，并可生成用以表达场景的 NavMesh。其中，AI 主体对象可获取至目标位置的最优路径。本章所涉及的 Unity 项目包含 4 个场景，读者可在 Unity 中打开对应项目，查看其工作方式，并以此了解所需构建的内容。通过此类项目示例，本节将学习如何创建 NavMesh，并在场景中与 AI 主体对象结合使用。

4.3.1 构建地图

下面首先构建相对简单的场景，如图 4.19 所示。

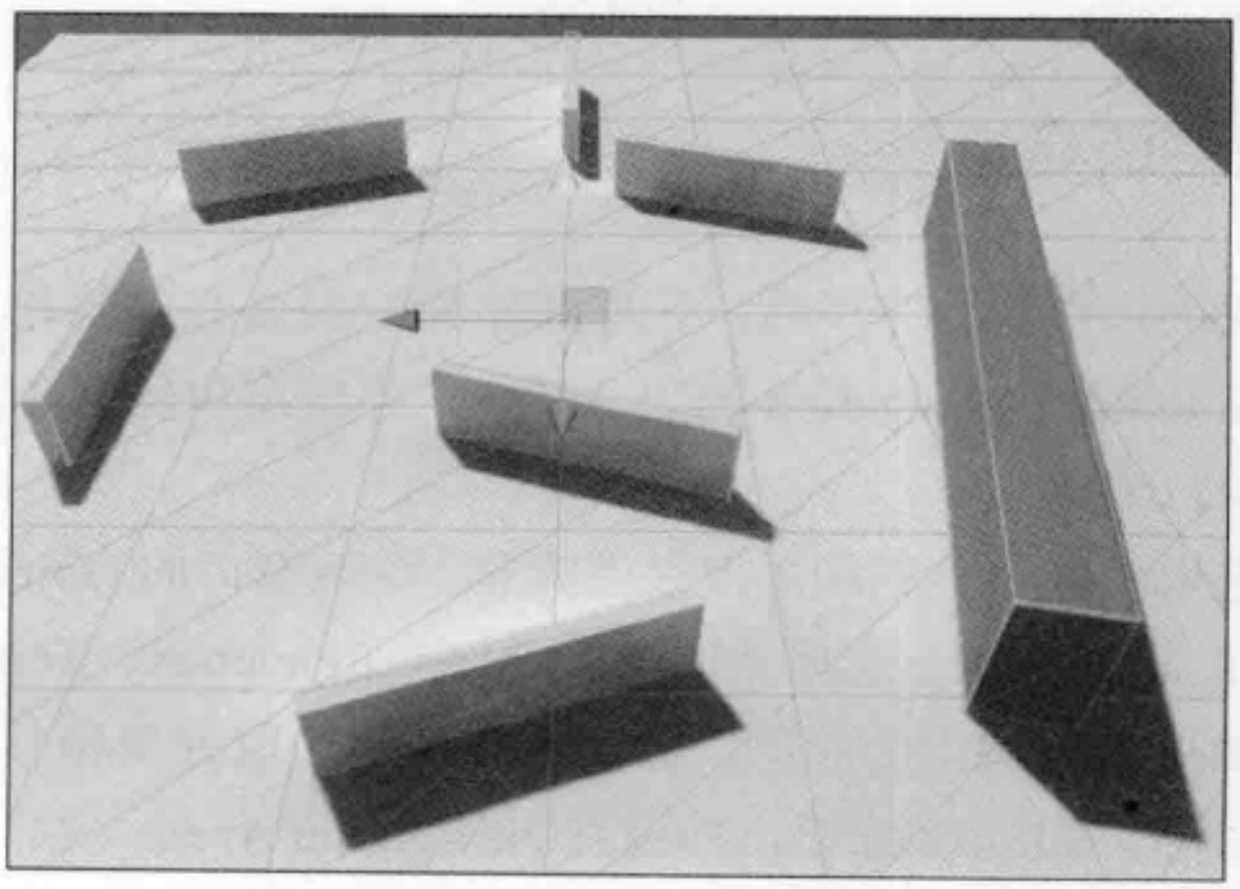

图 4.19

该场景名为 NavMesh01-Simple.scene，且位于本书的示例项目中。其中，可将平面用作地面对象，多个立方体实体则表示为墙体对象。稍后，场景中还将放置多个 AI 主体对象（坦克对象），其运动方式可通过鼠标点击位置加以操控，这与 RTS（即时战略）游戏十分类似。

4.3.2　静态障碍物

当加入了墙体和地面对象后，需要将其标记为 Navigation Static，以使 NavMesh 生成器了解到，此类对象表示为需要躲避的静态对象。当构建 NavMesh 时，仅需考察标记为 Navigation Static 的游戏对象，因而确保周边环境对象已被标记。对此，可选取全部对象，单击 Static 下拉列表，并选择 Navigation Static 选项，如图 4.20 所示。

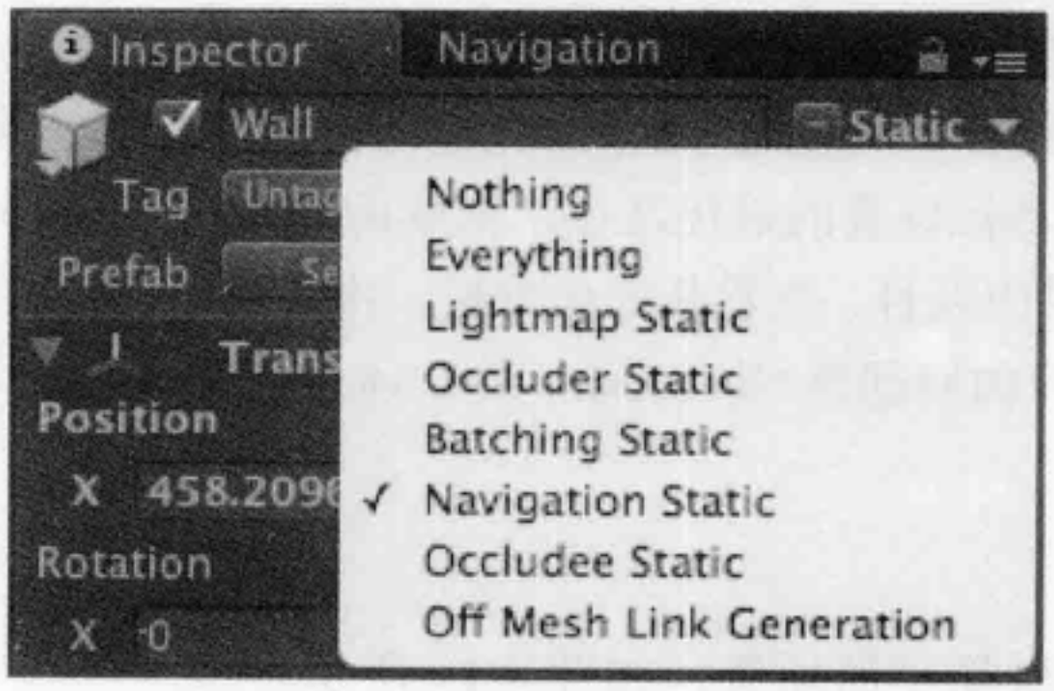

图 4.20

4.3.3　导航网格的烘焙

当与场景相关的操作执行完毕后，下面将对 NavMesh 仅需烘焙。首先需要打开导航窗口，并选择 Window | Navigation 选项。其中，导航窗口包含 3 个不同部分，首先是 Object，如图 4.21 所示。

针对对象的选取及其与导航相关属性的调整，导航窗口的 Object 选项卡可视为一种快捷方式。当在 Scene Filter 选项（即 All、Mesh Renderers、Terrains）之间进行切换时，将会过滤掉层次结构中的对象，因而可方便地选取并调整其 Navigation Static 标记和 Generate OffMeshLinks 标记，并设置 Navigation Area。

第二个选项卡为 Bake，如图 4.22 所示。

图 4.21

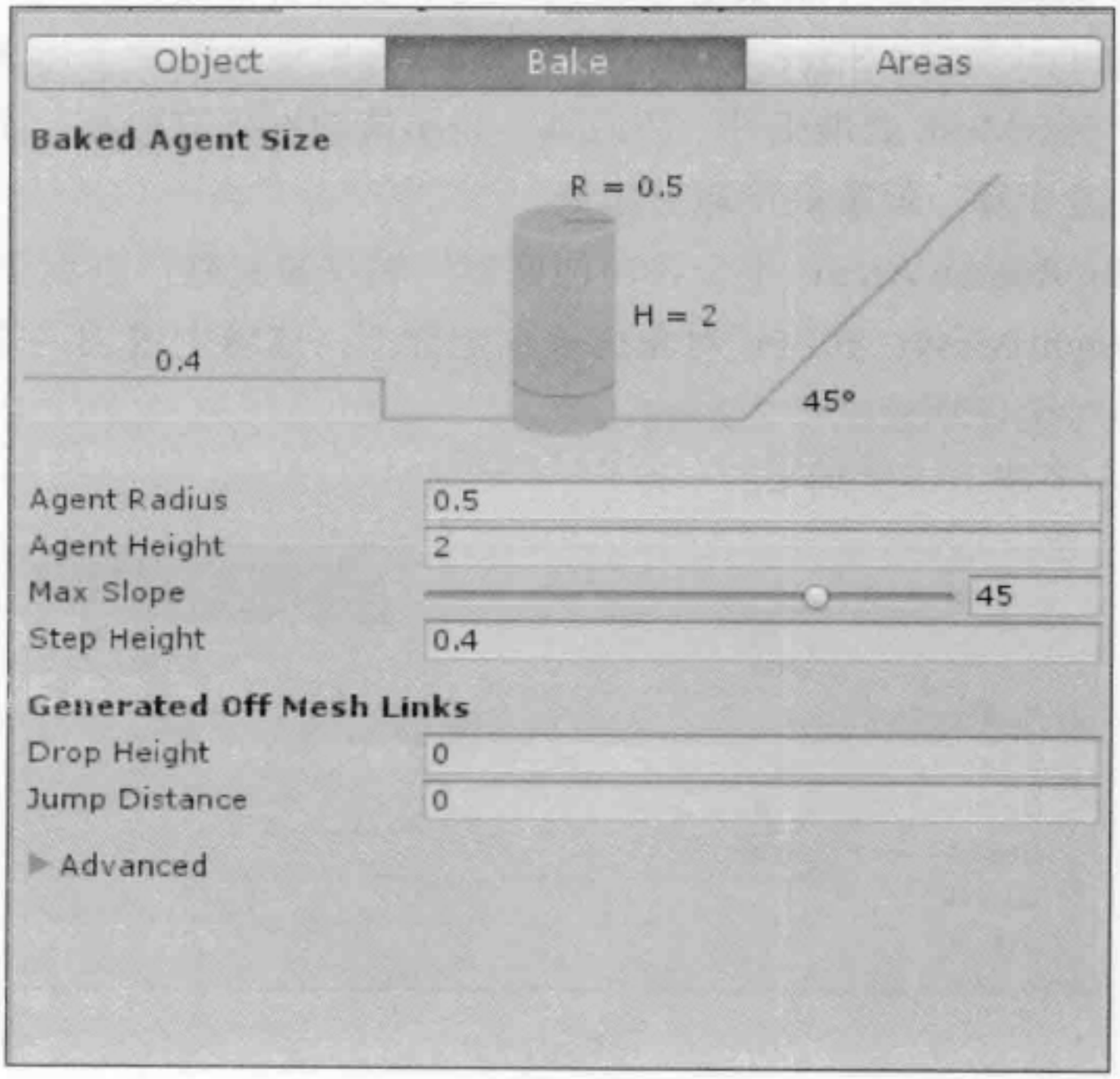

图 4.22

如果读者在 Unity 5 之前曾对该选项卡有所尝试，则会注意到当前选项卡稍显不同。Unity 5 增加了可视化工具，进而可准确地查看各项设置。下面将对此进行逐一考察。

- Agent Radius：Unity 文档将其描述为 NavMesh 主体对象的“个人空间”。主体对象需要躲避其他对象时，将会使用到该半径值。

- Agent Height：除了指定了主体对象的高度，进而判断是否可从障碍物下方通过之外，该项与 Agent Radius 类似。
- Max Slope：该项表示为主体对象可上行的最大角度值，对应坡度不可超过该值。
- Step Height：主体对象跨越的障碍物尺寸不得超出该值。

当构建 NavMesh 时，若选取了 GenerateOffMeshLinks，将使用到第二个数值分类选项。也就是说，主体对象可在其中行进，即使该过程中存在间隙。

- Drop Height：该项较为直观，即主体对象可向下跳跃的距离。例如，悬崖的高度。
- Jump Distance：该项表示为主体对象在分离网格链接之间跳跃的距离。

第三个参数集通常不需要进行调整，其中包括以下几个。

- Manual Voxel Size：Unity 中的 NavMesh 实现取决于体素，该设置可提升 NavMesh 的准确性。相应地，较小值的准确度较高，而较大值对应于快速计算，但准确度相对较低。
- Min Region Area：小于该项的面积区域将被剔除，进而予以忽略。
- Height Mesh：在主体对象的垂直定位上，这将生成较高的细节级别，但会对运行期速度产生影响。

图 4.23 显示了第三个选项卡。

图 4.23

回忆一下，Object 选项卡可将特定对象赋予某一区域，例如草皮、沙地等。随后，可将区域遮罩（mask）赋予主体对象，以使其可穿越或者无法穿越该区域。另外，估值参数将对主体对象穿越该区域的概率产生一定的影响。可能的话，主体对象通常会选择较低估值的路径。

当前示例相对简单，但读者可据此尝试各种设置。当前材质仅采用默认值，因而

读者仅需单击窗口下方的 Bake 按钮即可。针对当前场景，随后可看到烘焙 NavMesh 的进度条。最终，场景中将显示如图 4.24 所示的 NavMesh。

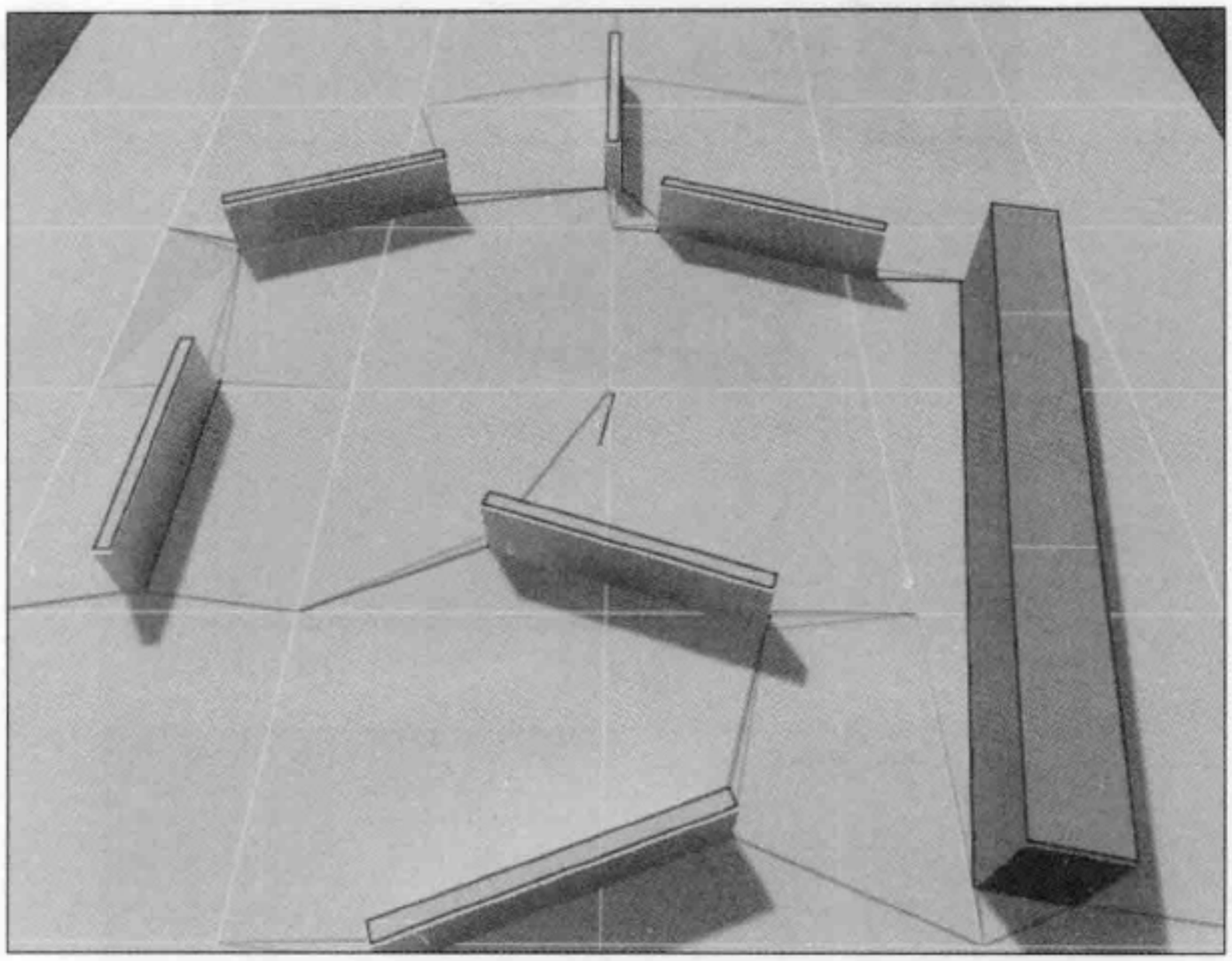

图 4.24

4.3.4 使用 NavMesh 主体对象

针对上述简单场景，当前材质已经完成了大部分工作，下面将向其添加某些 AI 主体对象，并以此查看是否可正常工作。此处采用了坦克模型，如图 4.25 所示。当然，读者也可将立方体或球体实体作为主体对象，其工作方式并无变化。

下一步是向坦克实体添加 Nav Mesh Agent 组件，该组件可简化路径搜索过程。由于 Unity 在后台处理寻路机制，因而用户无须直接对其进行操作。当在运行期内设置了组件的 destination 属性后，AI 主体对象自身将自动搜索路径。

相应地，可选择 Component | Navigation | Nav Mesh Agent 并添加该组件，如图 4.26 所示。

技巧：关于 Nav Mesh Agent，读者可访问 http://docs.unity3d.com/Documentation/Components/class-NavMeshAgent.html，进而参考与此相关的 Unity 文档。

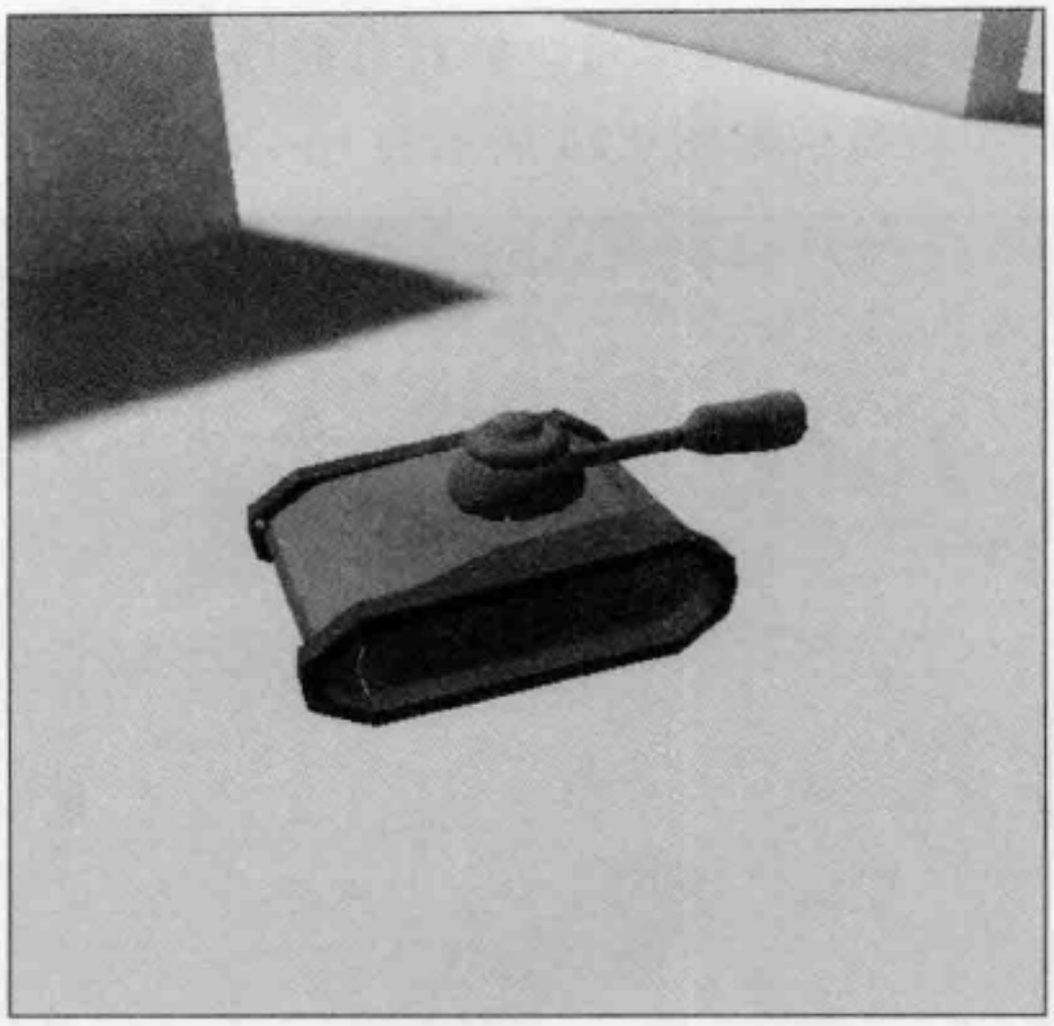

图 4.25

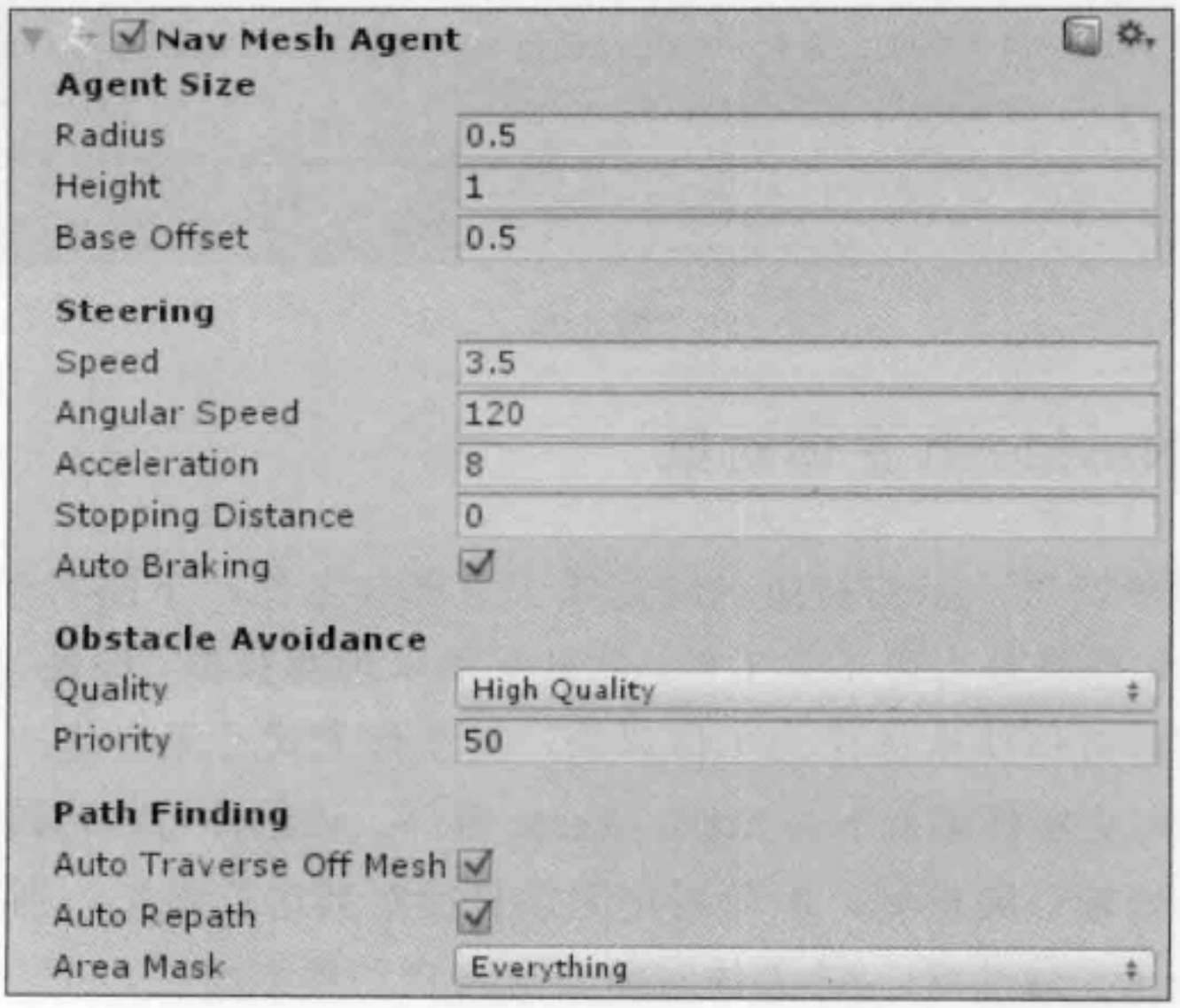

图 4.26

4.3.5　设置目的地

前述内容构建了 AI 主体对象，但需要一种方式以告知该对象的行进目标，并将

坦克的目标位置更新为鼠标点击位置。

对此，可使用球体实体作为标识对象，并于随后将 Target.cs 脚本绑定至某一空游戏对象上。接下来，将该球体实体拖曳至查看器中脚本的 targetMarker 转换属性上。

4.3.6 Target 类

Target 类较为简单并执行下列 3 项工作：

- 通过一条光线获取鼠标的点击位置。
- 更新标识位置。
- 更新全部 NavMesh 主体对象的 destination 属性。

Target 类的对应代码如下所示：

```
using UnityEngine;
using System.Collections;

public class Target : MonoBehaviour {
  private NavMeshAgent[] navAgents;
  public Transform targetMarker;

  void Start() {
    navAgents = FindObjectsOfType(typeof(NavMeshAgent)) as
        NavMeshAgent[];
  }

  void UpdateTargets(Vector3 targetPosition) {
    foreach (NavMeshAgent agent in navAgents) {
      agent.destination = targetPosition;
    }
  }

  void Update() {
    int button = 0;

    //Get the point of the hit position when the mouse is
    //being clicked
```

```
        if(Input.GetMouseButtonDown(button)) {
          Ray ray = Camera.main.ScreenPointToRay(
              Input.mousePosition);

          RaycastHit hitInfo;

          if (Physics.Raycast(ray.origin, ray.direction,
              out hitInfo)) {
            Vector3 targetPosition = hitInfo.point;
            UpdateTargets(targetPosition);
            targetMarker.position = targetPosition +

                new Vector3(0,5,0);
          }
        }
      }
    }
```

在游戏开始阶段，将在场景中搜索全部 NavMeshAgent 类型实体，并将其存储于 NavMeshAgent 索引数组中。当产生鼠标点击事件时，可执行简单的光线投射操作，并判断与该光线碰撞的首个对象。如果光线与某一对象相交，则更新标识的位置，同时更新各个 NavMesh 主体对象的目标位置，即利用新位置设置 destination 属性。本章将使用该脚本通知 AI 主体对象的目标位置。

最后，可对场景进行测试，并点击某一点并确定坦克的行进位置。坦克在躲避静态障碍物（例如墙体）的同时将逐渐接近目标点。

4.3.7 斜面测试

图 4.27 显示了包含某些斜面的场景。

需要注意的是，斜面和墙体之间应彼此连接。当在场景创建此类结合位置时，由于需要在后续操作中生成 NavMesh，因而对象间需要实现完美连接。否则，NavMesh 中将包含缝隙，主体对象将无法获取有效路径。斜面的连接状态如图 4.28 所示。

根据场景中主体对象行进时的斜面程度，可在 Navigation 窗口的 Bake 选项卡中调整 Max Slope 属性。此处使用了 45°，较大的 Max Slope 将生成更为陡峭的斜面。

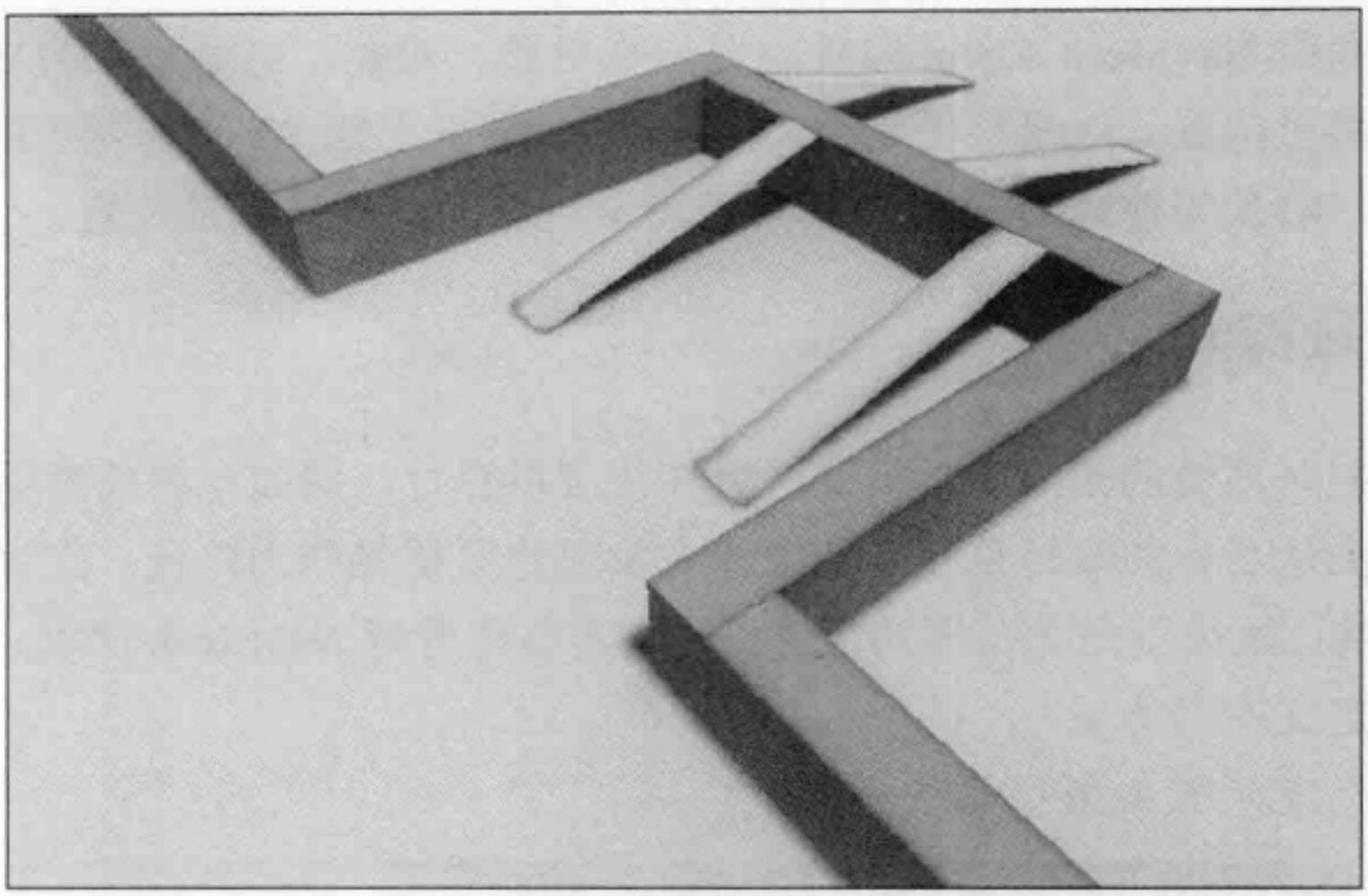

图 4.27

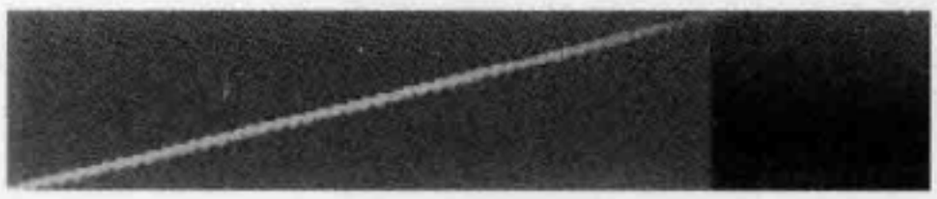

图 4.28

烘焙当前场景，最终将得到如图 4.29 所示的 NavMesh。

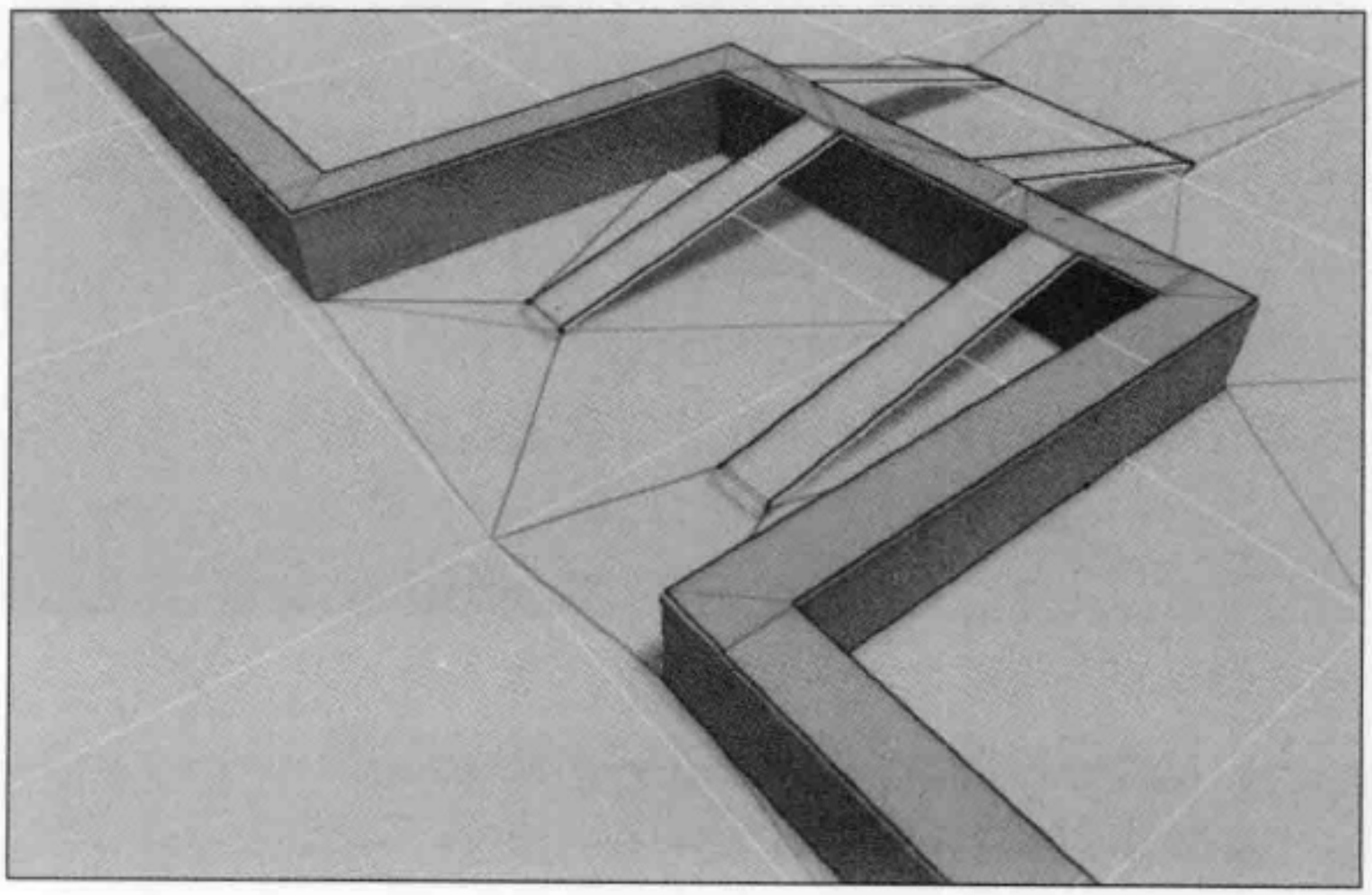

图 4.29

斜面将利用 NavMesh Agent 组件设置坦克对象。对此，可生成新的立方体对象，并用于目标参考位置。这里将采用前述 Target.cs 脚本更新 AI 主体对象的 destination 属性。最后，测试当前场景，AI 主体对象将穿越斜面并到达目标位置。

4.3.8　区域探索

在包含复杂环境的游戏中，某些区域往往难以穿行，例如池塘或湖面。尽管横渡池塘往往是到达目标的最短路径，但主体对象应选取桥梁作为路径。换言之，与取道桥梁相比，绕道池塘则是大范围的行进过程。本节将考察 NavMesh 区域，并采用不同的行进估值定义不同的层。

本节将构建如图 4.30 所示的场景。

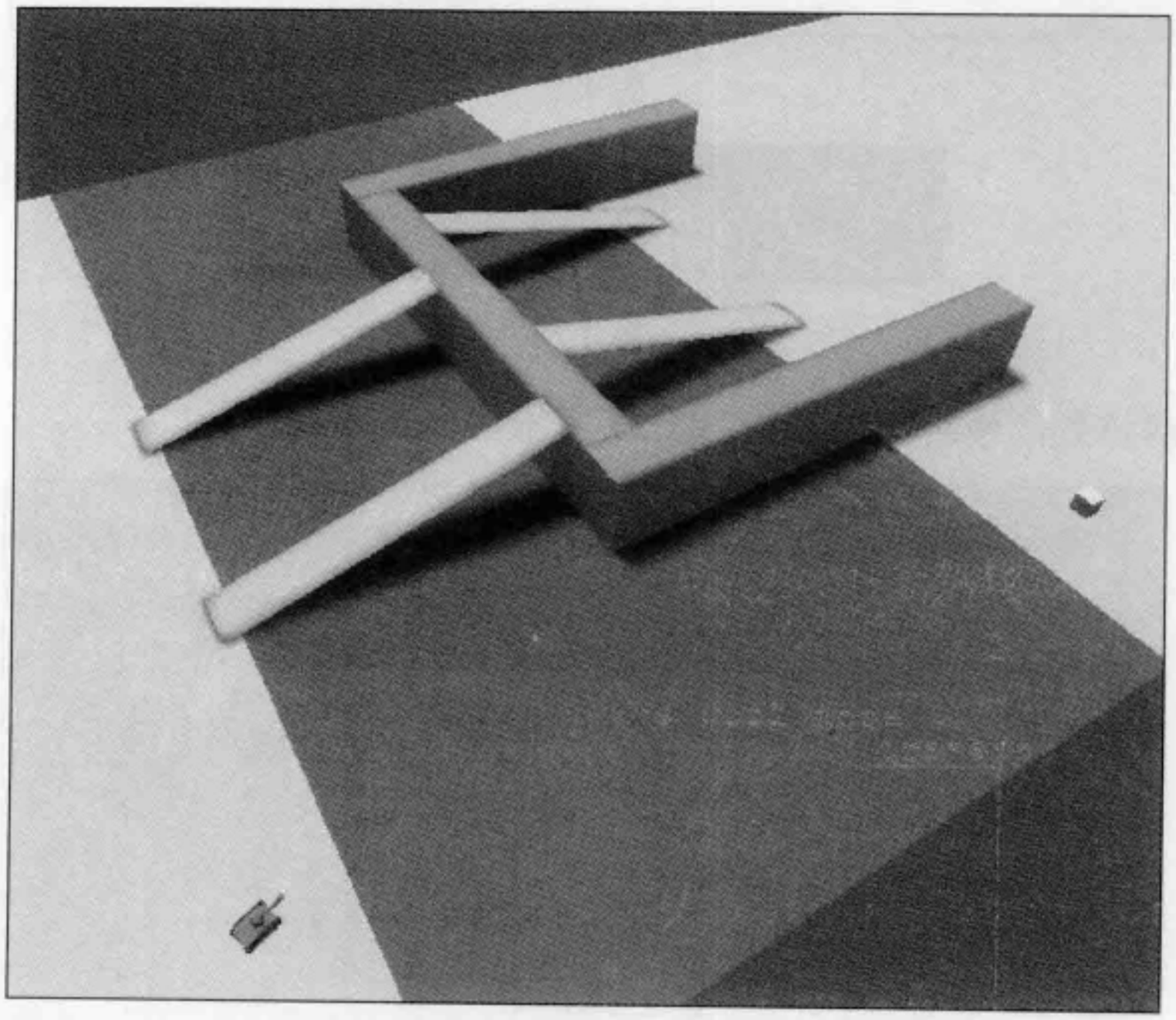

图 4.30

其中包含了 3 个平面表示两个由桥梁连接的地表平面，以及二者间的水域平面。通过观察可知，到达立方体处的最短路径为横渡水面，但此处希望 AI 主体对象选择桥梁，仅在必要时方可横渡水面区域，例如当目标对象位于水面上时。

图 4.31 显示了场景层次结构。当前游戏关卡由平面、斜面和墙体构成，且坦克实体以及目标立方体均与 Target.cs 脚本绑定。

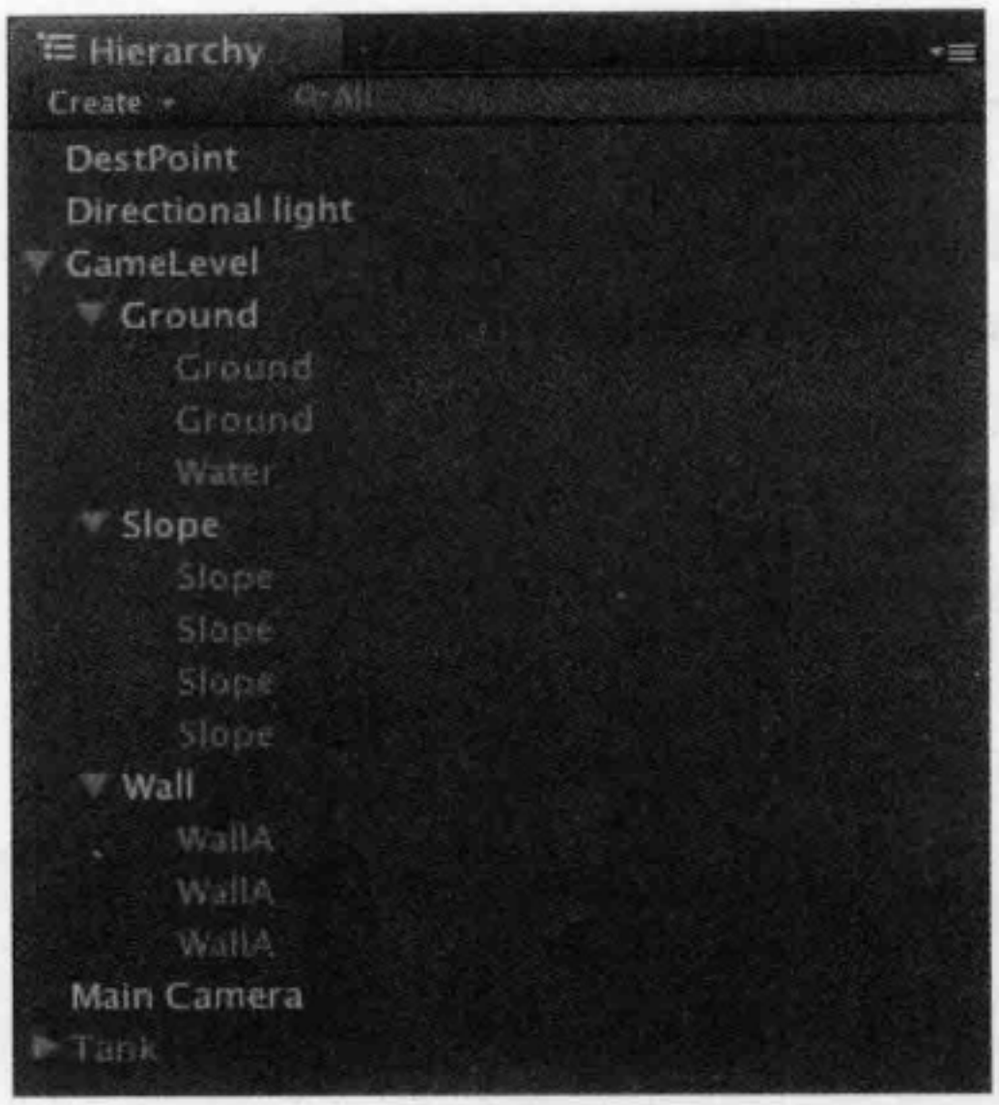

图 4.31

如前所述，NavMesh 可在 Navigation 窗口中的 Areas 选项卡中进行编辑。

Unity 包含了 3 个默认层，即 Default、Not Walkable 以及 Jump，各层均包含不同的估值。下面添加新层且对应估值为 5。

随后，选取水面平面，即打开 Navigation 窗口，在 Object 选项卡下将 Navigation Area 设置为 Water，如图 4.32 所示。

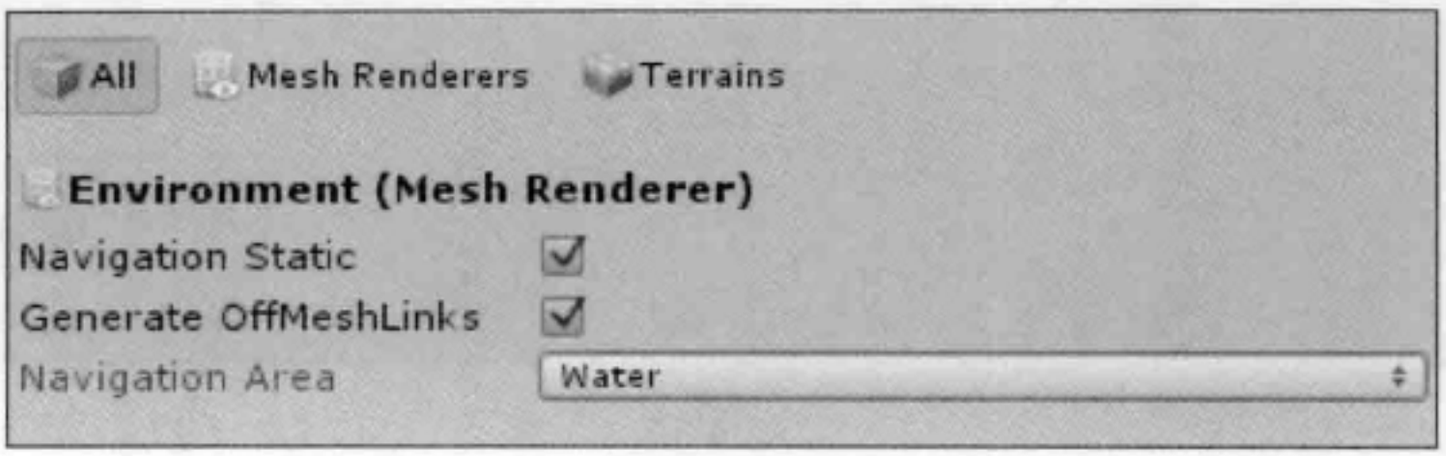

图 4.32

针对当前场景烘焙 NavMesh，并对其进行测试。此时，AI 主体对象将选取斜面，而非标记为水面层的平面——该行为代价相对高昂。另外，用户还可尝试将目标对象

置于水面平面上的不同点处。不难发现，AI 某些时候将游回岸边，并选取桥梁路径，而非横渡水面。

4.3.9 Off Mesh Links 连接

某些时候，场景可能存在缝隙，使得导航网格处于非连接状态。例如，在前述示例中，如果斜面与墙体之间未实现较好的连接，则主体对象将无法获取有效的路径。或者可设置对应点，以使主体对象可跳下墙体，并落入至下方平面上。Unity 包含 Off Mesh Links 特性，并可连接此类缝隙。Off Mesh Links 可通过手工方式构建，或通过 Unity 的 NavMesh 生成器自动生成。

图 4.33 显示了即将构建的示例场景，通过观察可知，两个平面间存在缝隙，下面考察如何通过 Off Mesh Links 连接两个平面。

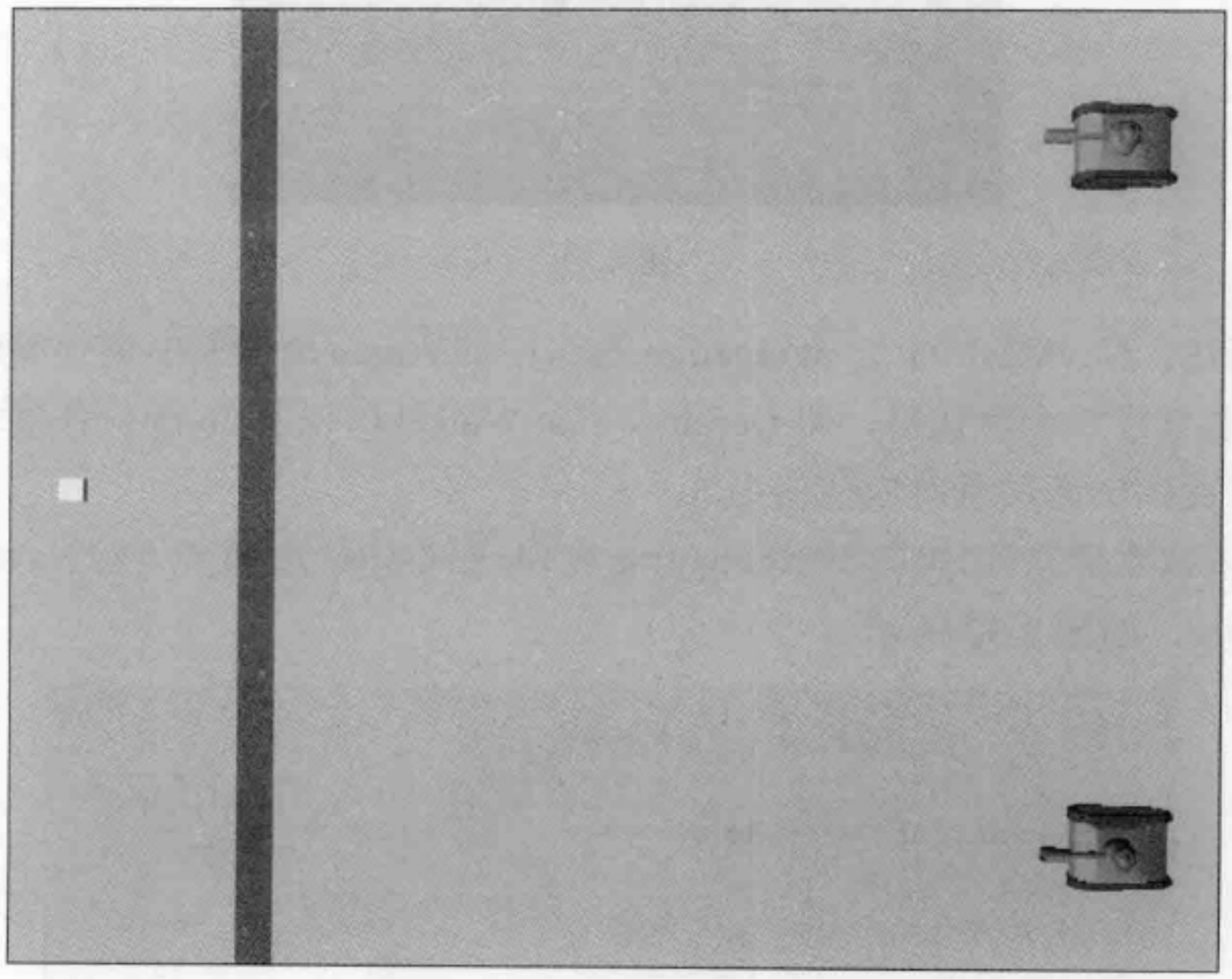

图 4.33

4.3.10 生成 Off Mesh Links

本节将通过自动方式生成 Off Mesh Links，进而连接两个平面。首先可将这两个

平面在树形查看器中标记为 Off Mesh Link Generation，如图 4.34 所示。

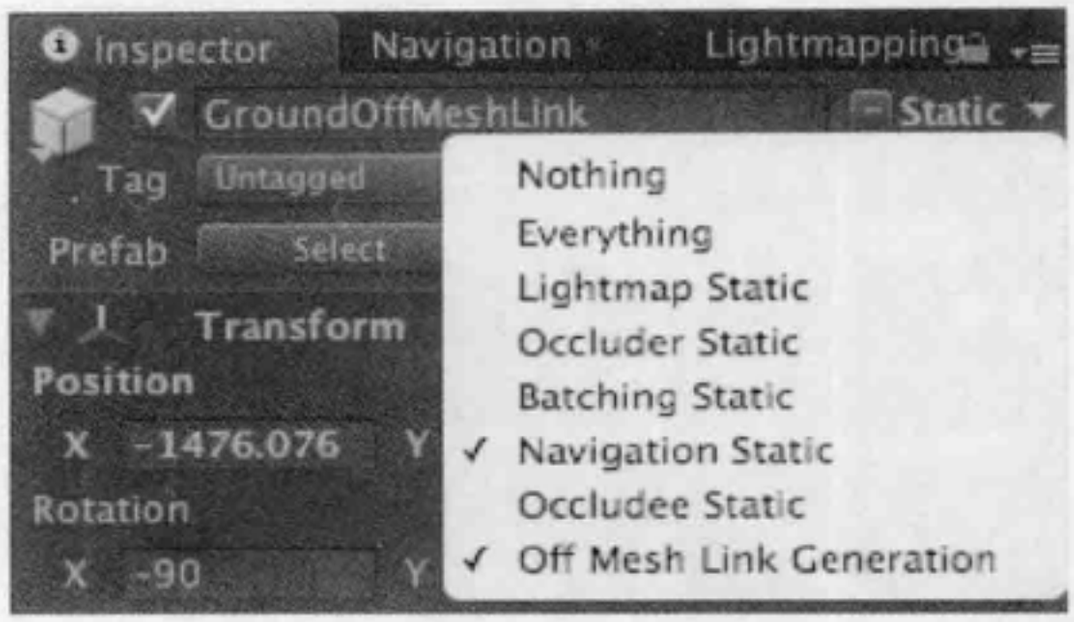

图 4.34

用户可设置距离阈值，并在 Navigation 窗口的 Bake 选项卡中自动生成 Off Mesh Links。

点击 Bake 后，Off Mesh Links 将按照如图 4.35 所示的方式连接两个平面。

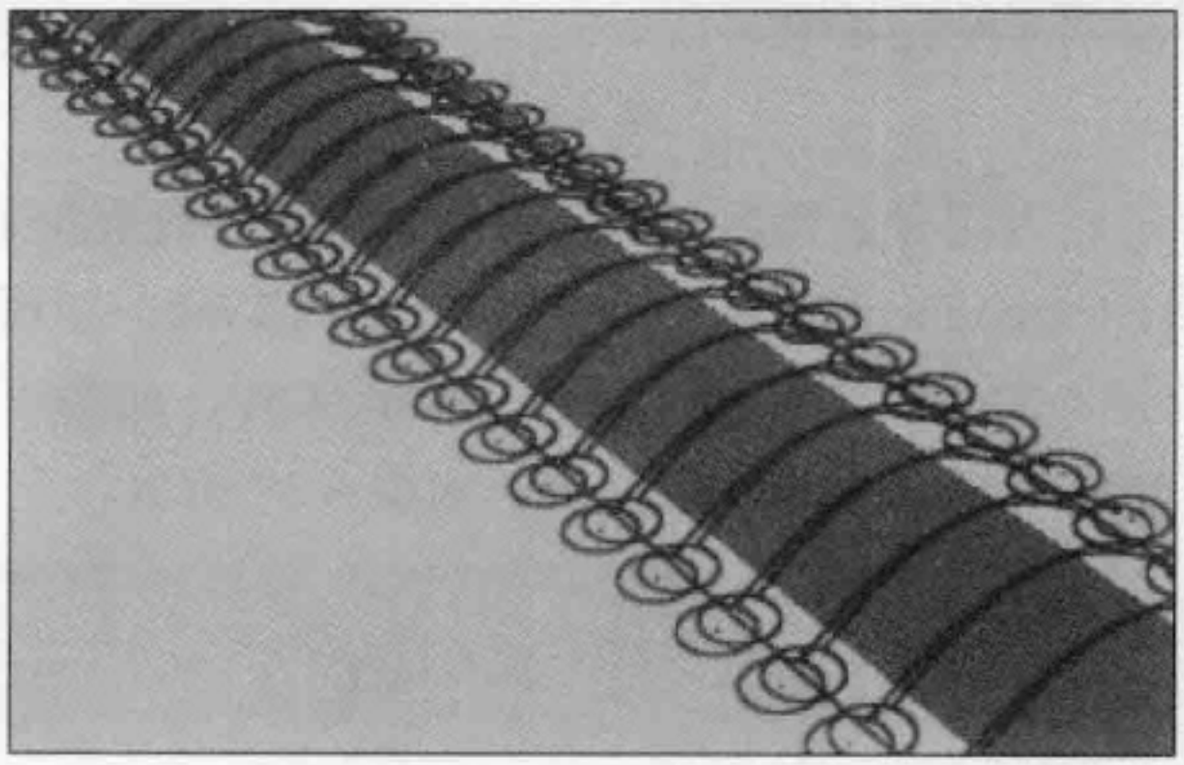

图 4.35

当前，AI 主体对象可获取有效路径穿越两个平面。当到达平面边缘并发现 Off Mesh Link 后，主体对象将直接“瞬移”至另一平面处。除非特殊强调这一类“瞬移”主体对象，否则应架设一座桥梁以使主体对象顺利通过。

4.3.11 设置 Off Mesh Links

如果不打算沿平面边缘生成 Off Mesh Links，并希望主体对象直接“瞬移”至另

一平面上的某一点处，如图 4.36 所示，还可通过人工方式设置 Off Mesh Links。

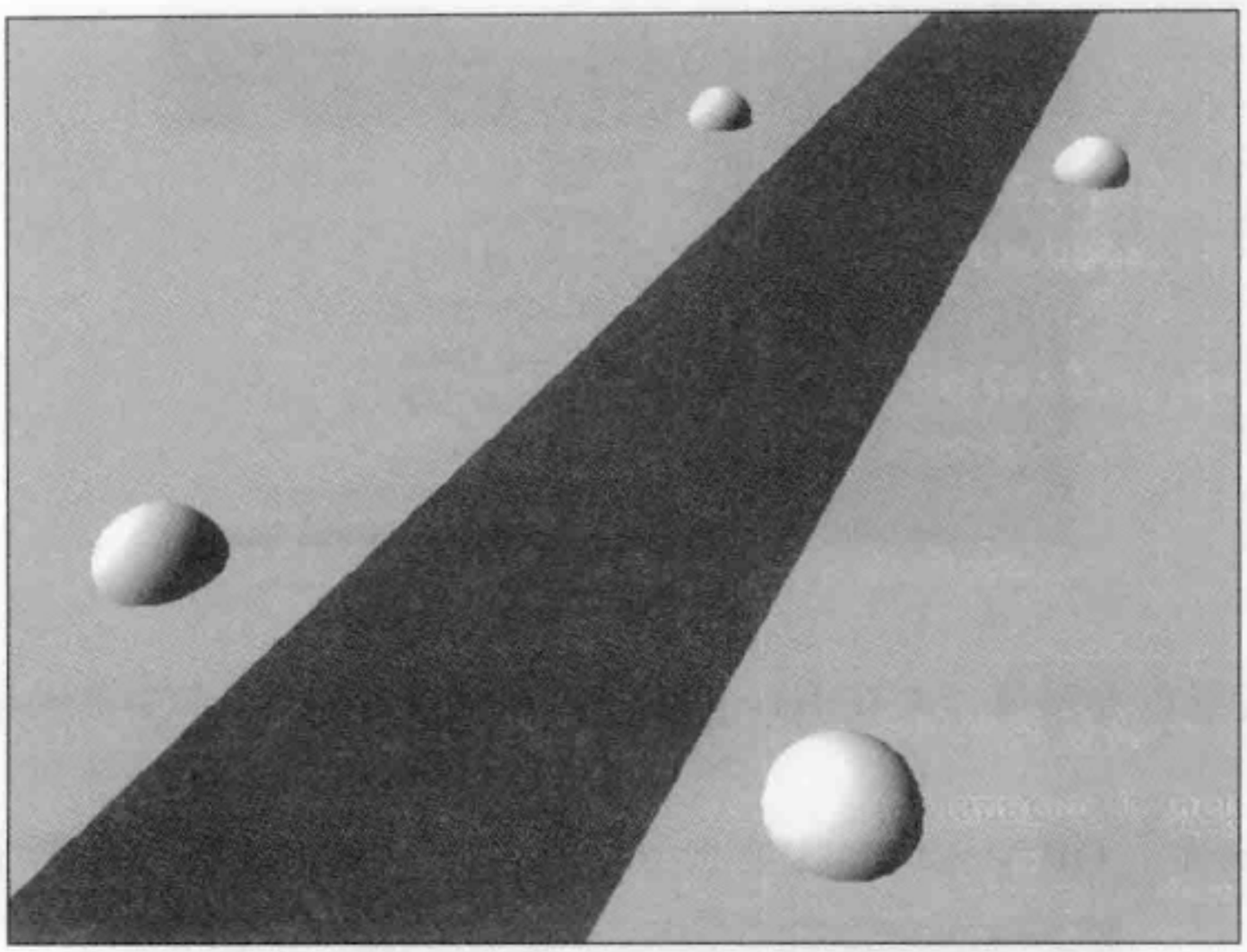

图 4.36

图 4.36 中包含了较为明显的缝隙，并且在平面两侧分别放置了两组球体。选取某一球体对象，并通过 Component | Navigation | Off Mesh Link，即可向其添加 Off Mesh Links。需要说明的是，此处仅需向一个球体添加该组件。随后，可将该球体拖曳至 Start 属性中，其他球体则拖曳至 End 属性中，如图 4.37 所示。

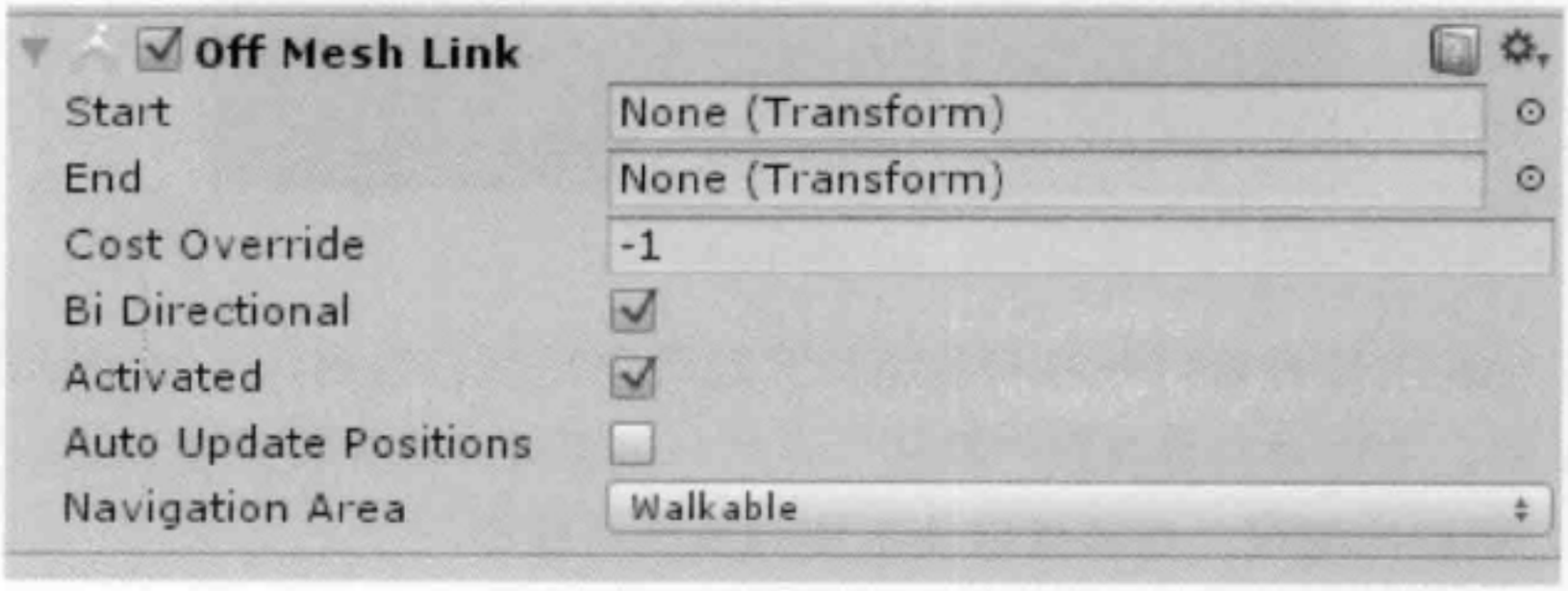

图 4.37

接下来，进入 Navigation 窗口并烘焙当前场景。当前，平面通过人工 Off Mesh Links 予以连接，AI 主体对象可穿越图 4.38 中所示的缝隙。

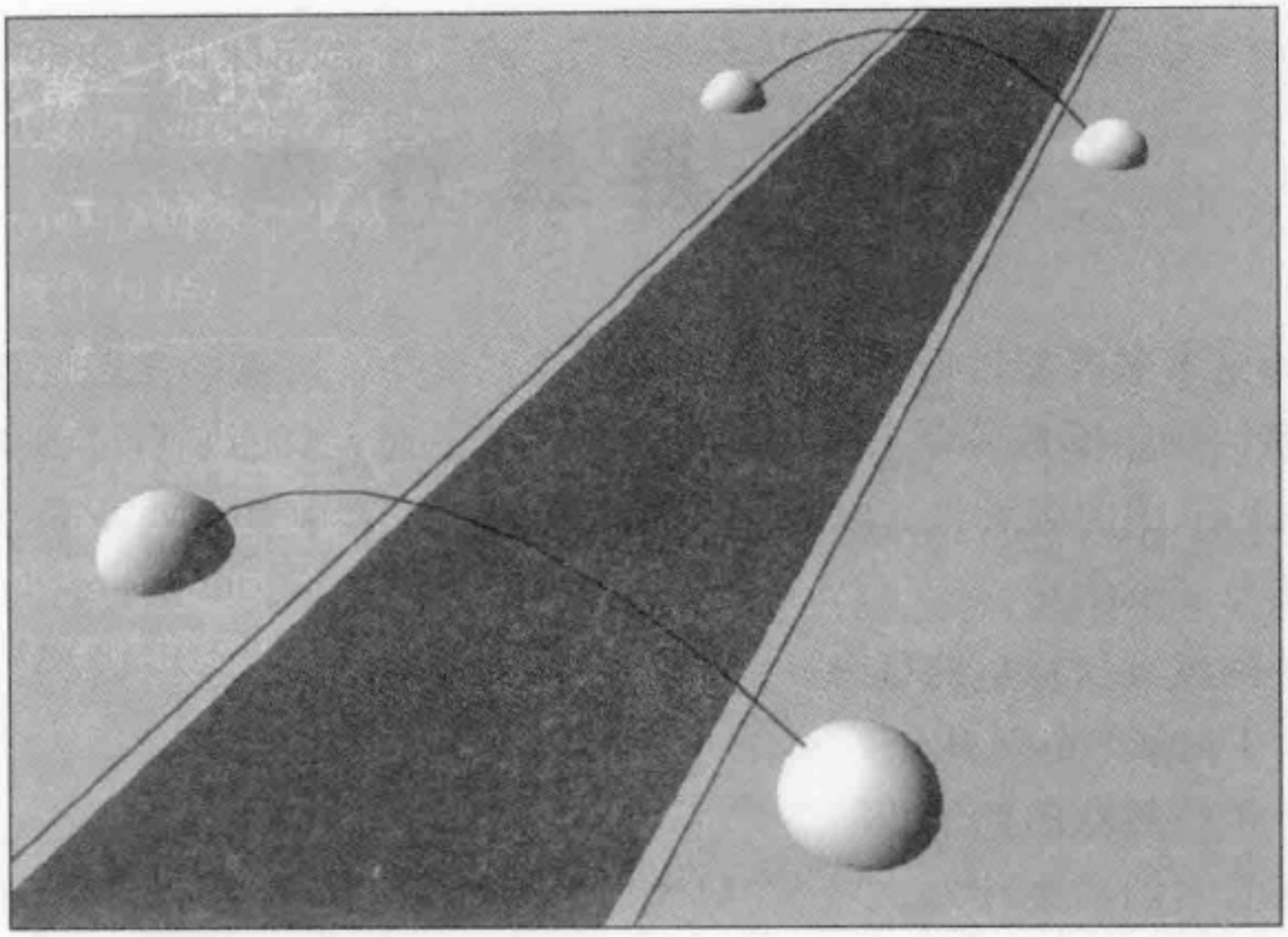

图 4.38

4.4 本章小结

本章首先讨论了基本的路点系统，随后是简单的 A*寻路系统的实现方法，最后介绍了 Unity 内建的导航系统。多数内容采用了 Unity 中相对简洁的 NavMesh 系统，其他部分则使用了更为详细的 A*定制实现方法。更为重要的是，读者应了解上述不同系统的使用时机和方式。

尽管读者尚未意识到，但前述内容已经通过其他概念实现了上述不同系统间的整合操作。

第 5 章将讨论群集算法，并进一步扩展本章所涉及的诸多概念，进而以高效、可信方式实现主体对象的群集运动行为。

第5章　群集行为

群集行为是本书所要讨论的另一个基本概念，其中，鸟群的实现过程相对简单，并可通过少量代码向模拟环境中添加大量的真实感；而人群的实现过程则稍显复杂，本章将通过 Unity 中的强大工具实现这一模拟行为。本章主要涵盖以下主题：

- 群集算法的历史。
- 理解群集算法背后的概念。
- 基于 Unity 的群集算法。
- 基于传统算法的群集算法。
- 使用真实的群集行为。

5.1　群集算法初探

群集算法可追溯到 20 世纪 80 年代中期，该算法由 Craig Reynolds 首先提出，并应用于电影工业，即于 1992 年拍摄的《蝙蝠侠归来》，其中模拟了蝙蝠的群集运动行为。该技术为 Craig Reynolds 赢得了奥斯卡奖项。随后，群集算法也涉足于其他不同领域，包括游戏业以及科研领域。尽管群集算法具有高效、准确等特征，但其理解和实现方式并不复杂。

5.2　理解群集算法背后的概念

类似于之前讨论的概念，理解群集算法的简单方式是将其与真实行为所联系。初看之下，此类概念描述了一组对象的整体运动方式。具体而言，鸟群算法根据自然界中鸟类的飞行行为而得来。期间，飞鸟朝某一目的地行进，且个体间保持固定距离，此处重点强调群体性行为。另外，本章还将讨论个体的运动和决策方式。相比较而言，鸟群算法则对大量的主体对象群体进行模拟，在对个体 boid 建模的同时实现整体方式运动。其中，各 boid 并不依赖于随机性和预定义路径。

本章将实现不同版本的群集算法。其中，第一个算法源自 Tropical Paradise Island 示例项目中的群集行为。该示例项目位于 Unity 2.0 中，但在 Unity 3.0 及其后续版本中被移除。对于首个示例，本章将对应代码用于 Unity 5 中。第二个算法基于 Craig Reynold 群集算法，读者可查看二者间的差异和相似性。其中定义了与群集工作方式相关的 3 个基本概念，自 20 世纪 80 年代算法出现以来，此类概念一直被视为算法的主体内容，其中包括以下方面。

- 分离性：为了避免碰撞，个体间应保持固定的距离，如图 5.1 所示。

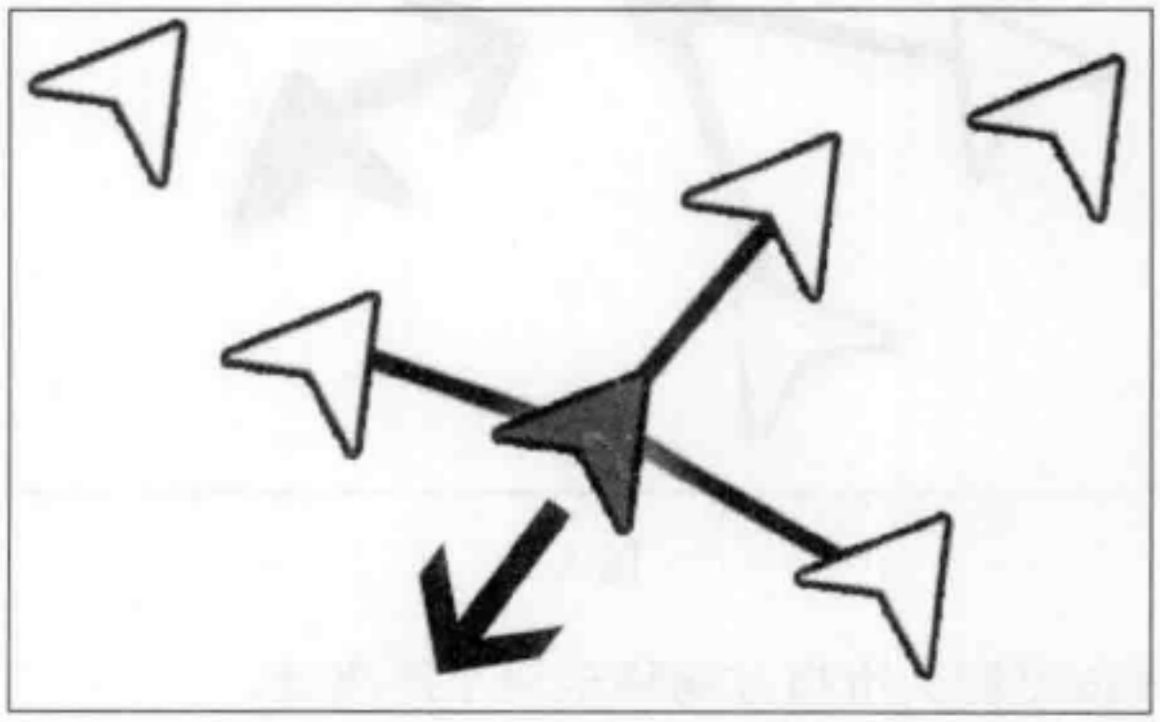

图 5.1

其中，中间个体在不改变飞行方向的前提下远离其他个体。

- 排列的整齐性：整体间以相同方向和速度运动，如图 5.2 所示。

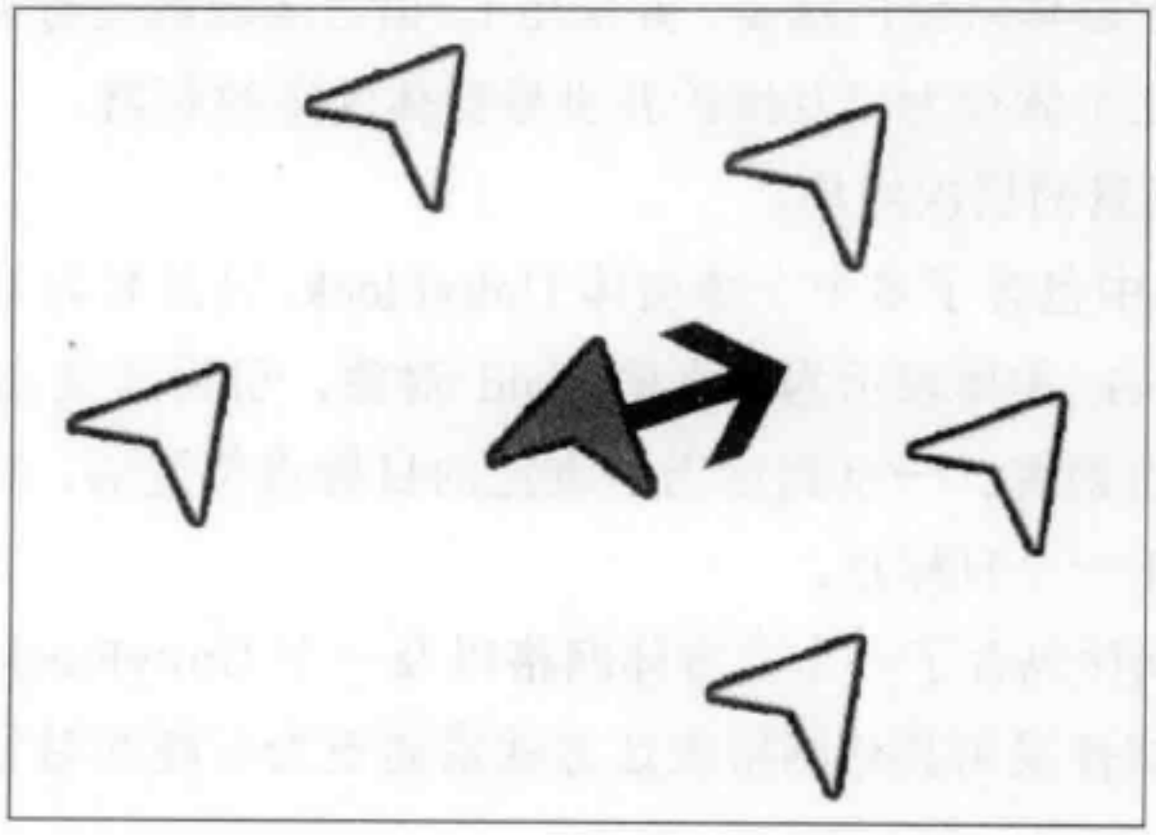

图 5.2

其中，中间个体按照箭头所指方向调整其行进路线，并与整体前进方向保持一致。

- 内聚性：与集群中心位置间保持最小距离，如图 5.3 所示。

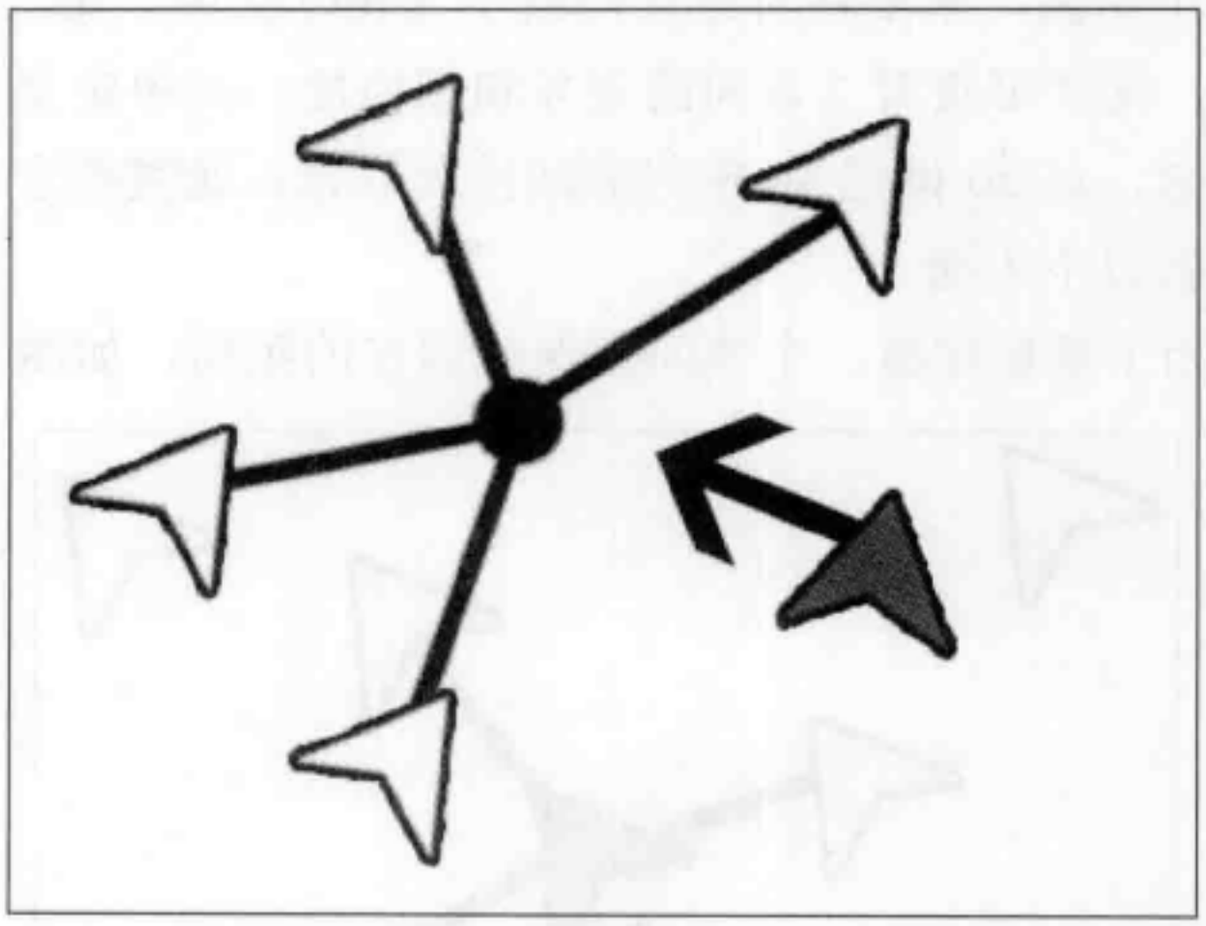

图 5.3

其中，右侧个体按照箭头所指方向移至最小距离处。

5.3　Unity 示例中的群集行为

本节将创建基于群体对象的场景，并采用 C#语言实现群集行为。该示例包含了两项主要内容，即独立个体行为以及维护并引导群体的主控制器。

图 5.4 显示了场景的层次结构。

不难发现，示例中包含了多个个体实体 UnityFlock，以及名为 UnityFlockController 的控制器。UnityFlock 实体表示为独立的 boid 对象，引用其父 UnityFlockController 实体，并将其视为引领者。一旦到达当前设定的目标点位置后，UnityFlockController 实体将随机更新至下一个目标点。

UnityFlock 预制件包含了一个立方体网格以及一个 UnityFlock 脚本。除此之外，读者还可针对该预制件采用其他网格表达方式描述更为有趣的对象，例如鸟类。

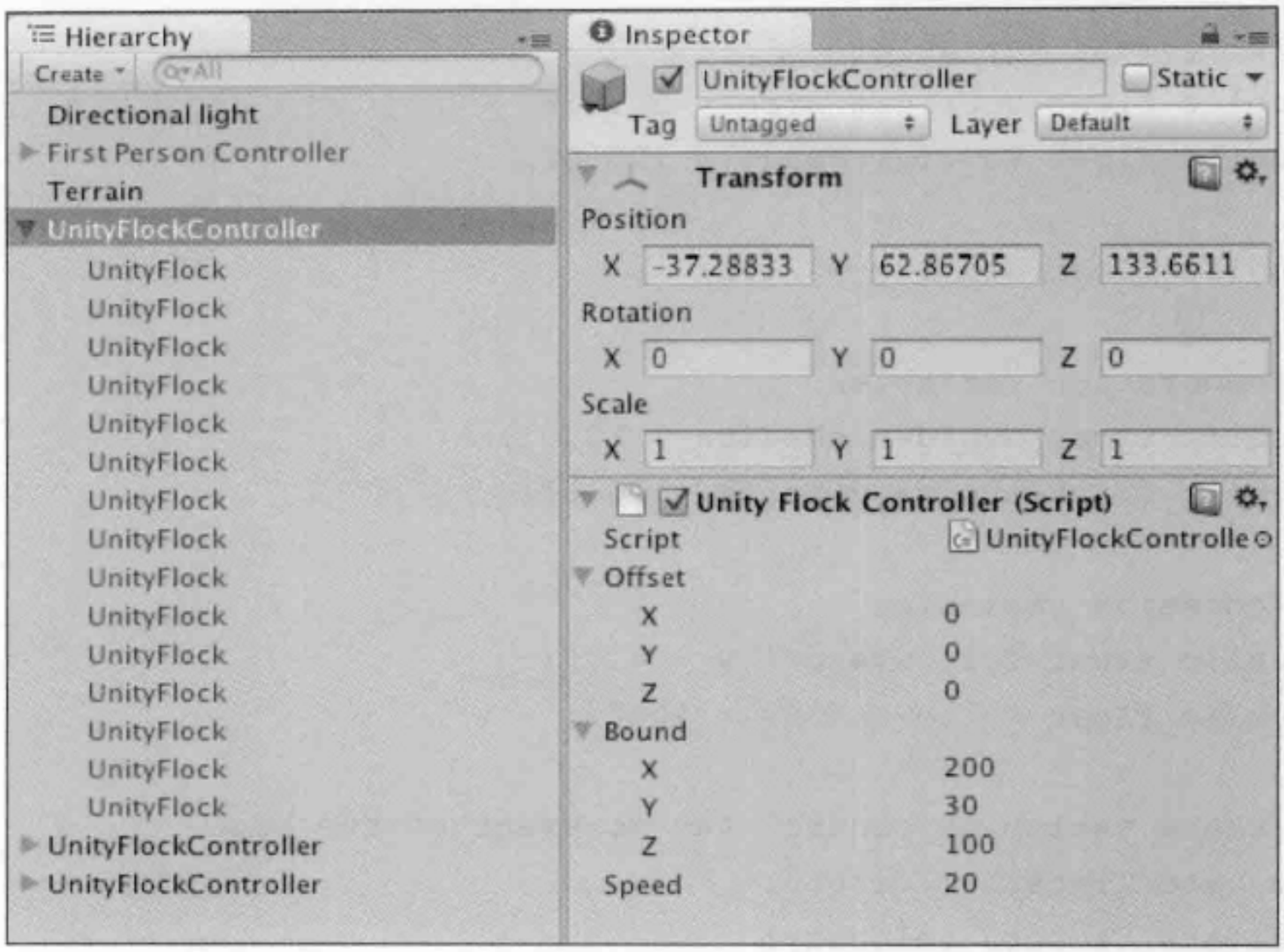

图 5.4

5.3.1 模拟个体行为

术语 boid 由 Craig Reynold 首次提出，用于描述类似于鸟类的对象，本节将采用该术语表示群集中的个体对象，并实现 boid 行为。UnityFlock.cs 脚本中的相关内容用于控制群集中的各个 boid。

UnityFlock.cs 文件中的示例代码如下所示：

```
using UnityEngine;
using System.Collections;

public class UnityFlock : MonoBehaviour {
  public float minSpeed = 20.0f;
  public float turnSpeed = 20.0f;
  public float randomFreq = 20.0f;
  public float randomForce = 20.0f;
```

```
//alignment variables
public float toOriginForce = 50.0f;
public float toOriginRange = 100.0f;

public float gravity = 2.0f;

//seperation variables
public float avoidanceRadius = 50.0f;
public float avoidanceForce = 20.0f;

//cohesion variables
public float followVelocity = 4.0f;
public float followRadius = 40.0f;

//these variables control the movement of the boid
private Transform origin;
private Vector3 velocity;
private Vector3 normalizedVelocity;
private Vector3 randomPush;
private Vector3 originPush;
private Transform[] objects;
private UnityFlock[] otherFlocks;
private Transform transformComponent;
```

代码中声明了算法所需的输入值，此类值可在编辑器中设置和定义。其中，首先定义了 boid 的最小运动速度 minSpeed 以及旋转速度 turnSpeed。randomFreq 值表示根据 randomForce 值的、更新 randomPush 值的次数。此处，所施加的作用力用于随机生成速度，进而使得群集的整体行为更具真实感。

toOriginRange 值定义了群集的展开方式；另外，toOriginForce 使得 boid 处于有效范围内，并与群集原点保持某一距离。基本上讲，此类属性可用于处理群集算法中的排列整齐性这一问题。avoidanceRadius 和 avoidanceForce 属性则用于维护 boid 个体间的最小距离，即群集算法中的分离规则。

followRadius 和 followVelocity 用于维护与引领者或群集原点位置间的最小距离，

这也体现了群集算法中的内聚性规则。

origin 对象表示为父对象，并控制群集的整体行为。鉴于当前 boid 需要了解群集中的其他 boid，因而可使用 objects 和 otherFlocks 属性存储邻近 boid 的信息。

boid 的初始化方法如下所示：

```
void Start () {
  randomFreq = 1.0f / randomFreq;

  //Assign the parent as origin
  origin = transform.parent;

  //Flock transform
  transformComponent = transform;

  //Temporary components
  Component[] tempFlocks= null;

  //Get all the unity flock components from the parent
  //transform in the group
  if (transform.parent) {
  tempFlocks = transform.parent.GetComponentsInChildren
     <UnityFlock>();
  }

  //Assign and store all the flock objects in this group
  objects = new Transform[tempFlocks.Length];
  otherFlocks = new UnityFlock[tempFlocks.Length];

  for (int i = 0;i<tempFlocks.Length;i++) {
    objects[i] = tempFlocks[i].transform;
    otherFlocks[i] = (UnityFlock)tempFlocks[i];
  }

  //Null Parent as the flock leader will be
  //UnityFlockController object
```

```
        transform.parent = null;

        //Calculate random push depends on the random frequency
    //provided
        StartCoroutine(UpdateRandom());
      }
```

此处将当前 boid 对象的父对象设置为原点，也就是说，其他个体需要跟从的控制器对象。随后，可获取群组中的其他 boid，并将其存储于变量中，以供后续操作引用。StartCoroutine 方法首先调用协同例程 UpdateRandom()方法，如下所示：

```
IEnumerator UpdateRandom() {
  while (true) {
    randomPush = Random.insideUnitSphere * randomForce;
    yield return new WaitForSeconds(randomFreq +
        Random.Range(-randomFreq / 2.0f, randomFreq / 2.0f));
  }
}
```

UpdateRandom()方法利用基于 randomFreq 的时间间隔更新游戏中的 randomPush 值。另外，Random.insideUnitSphere 部分返回 Vector3 对象，该对象包含了半径为 randomForce 的球体内的 x、y、z 值。随后，可等待某一随机时间值，恢复 while(true)循环并再次更新 randomPush 值。

针对前述内容讨论的群集算法的 3 项规则，boid 行为的 Update()方法实现了 boid 实体的对应内容，如下所示：

```
void Update () {
  //Internal variables
  float speed = velocity.magnitude;
  Vector3 avgVelocity = Vector3.zero;
  Vector3 avgPosition = Vector3.zero;
  float count = 0;
  float f = 0.0f;
  float d = 0.0f;
```

```
Vector3 myPosition = transformComponent.position;
Vector3 forceV;
Vector3 toAvg;
Vector3 wantedVel;

for (int i = 0;i<objects.Length;i++){
  Transform transform= objects[i];
  if (transform != transformComponent) {
    Vector3 otherPosition = transform.position;
//Average position to calculate cohesion
avgPosition += otherPosition;
count++;

//Directional vector from other flock to this flock
forceV = myPosition - otherPosition;

//Magnitude of that directional vector(Length)
d= forceV.magnitude;

//Add push value if the magnitude, the length of the
//vector, is less than followRadius to the leader
if (d < followRadius) {
  //calculate the velocity, the speed of the object, based
    //on the avoidance distance between flocks if the
  //current magnitude is less than the specified
  //avoidance radius
  if (d < avoidanceRadius) {
    f = 1.0f - (d / avoidanceRadius);
  if (d > 0) avgVelocity +=
      (forceV / d) * f * avoidanceForce;
}

//just keep the current distance with the leader
f = d / followRadius;
```

```
            UnityFlock tempOtherFlock = otherFlocks[i];
            //we normalize the tempOtherFlock velocity vector to get
            //the direction of movement, then we set a new velocity
            avgVelocity += tempOtherFlock.normalizedVelocity * f *
                followVelocity;
        }
    }
}
```

上述代码实现了分离规则。首先，代码检测当前 boid 与其他 boid 之间的距离并对速度予以更新，具体解释内容可参考代码中的注释。

接下来可计算群集的平均速度，即利用当前速度除以群集中的 boid 数量，如下所示：

```
if (count > 0) {
  //Calculate the average flock velocity(Alignment)
  avgVelocity /= count;

  //Calculate Center value of the flock(Cohesion)
  toAvg = (avgPosition / count) - myPosition;
}
else {
  toAvg = Vector3.zero;
}

//Directional Vector to the leader
forceV = origin.position - myPosition;
d = forceV.magnitude;
f = d / toOriginRange;

//Calculate the velocity of the flock to the leader
if (d > 0) //if this void is not at the center of the flock
    originPush = (forceV / d) * f * toOriginForce;

if (speed < minSpeed && speed > 0) {
```

```
        velocity = (velocity / speed) * minSpeed;
    }

    wantedVel = velocity;

    //Calculate final velocity
    wantedVel -= wantedVel * Time.deltaTime;
    wantedVel += randomPush * Time.deltaTime;
    wantedVel += originPush * Time.deltaTime;
    wantedVel += avgVelocity * Time.deltaTime;
    wantedVel += toAvg.normalized * gravity * Time.deltaTime;

    //Final Velocity to rotate the flock into
    velocity = Vector3.RotateTowards(velocity, wantedVel,
        turnSpeed * Time.deltaTime, 100.00f);

    transformComponent.rotation =
Quaternion.LookRotation(velocity);

    //Move the flock based on the calculated velocity
    transformComponent.Translate(velocity * Time.deltaTime,
        Space.World);

    //normalise the velocity
    normalizedVelocity = velocity.normalized;
  }
}
```

最后，可完善全部因素，例如 randomPush、originPush 以及 avgVelocity，进而计算最终的目标速度 wantedVel。另外，还可利用 Vector3.RotateTowards 方法中的线性插值技术将当前速度更新为 wantedVel。随后，可采用 Translate()方法以及最新速度移动 boid 对象。

下面将创建立方体对象，并向其中加入 UnityFlock 脚本，进而生成预制组件，如图 5.5 所示。

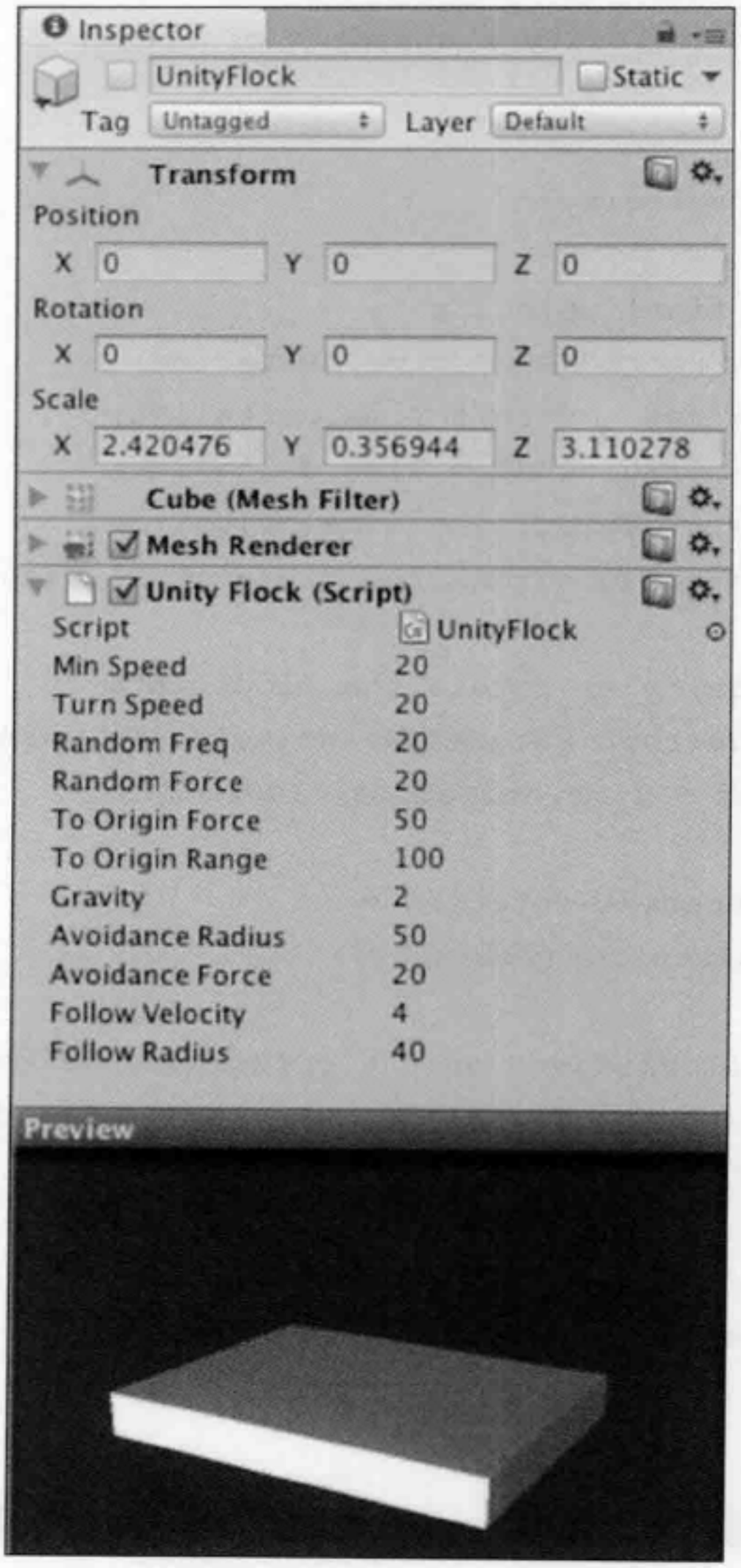

图 5.5

5.3.2　创建控制器

下面定义控制器类，该类更新其自身位置，以使其他 boid 对象了解整体行进方向。该变量引自前述 UnityFlock 脚本中的 origin 变量。

UnityFlockController.cs 文件中的对应代码如下所示：

```
using UnityEngine;
using System.Collections;

public class UnityFlockController : MonoBehaviour {
  public Vector3 offset;
  public Vector3 bound;
  public float speed = 100.0f;

  private Vector3 initialPosition;
  private Vector3 nextMovementPoint;

  //Use this for initialization
  void Start () {
    initialPosition = transform.position;
    CalculateNextMovementPoint();
  }

  //Update is called once per frame
  void Update () {
    transform.Translate(Vector3.forward * speed * Time.deltaTime);
    transform.rotation = Quaternion.Slerp(transform.rotation,
        Quaternion.LookRotation(nextMovementPoint -
        transform.position), 1.0f * Time.deltaTime);

    if (Vector3.Distance(nextMovementPoint,
        transform.position) <= 10.0f)
        CalculateNextMovementPoint();
  }
```

Update()方法负责检测控制器对象是否位于目标位置点附近。若是，则利用之前讨论的 CalculateNextMovementPoint()方法再次更新 nextMovementPoint 变量，如下所示：

```
  void CalculateNextMovementPoint () {
    float posX = Random.Range(initialPosition.x - bound.x,
        initialPosition.x + bound.x);
```

```
        float posY = Random.Range(initialPosition.y - bound.y,

            initialPosition.y + bound.y);
        float posZ = Random.Range(initialPosition.z - bound.z,
            initialPosition.z + bound.z);

        nextMovementPoint = initialPosition + new Vector3(posX,
            posY, posZ);
    }
}
```

在当前位置和边界向量之间的范围内，CalculateNextMovementPoint()方法获取下一个随机目标位置。

综上所述，群集对象可获得较为真实的运动行为，如图 5.6 所示。

图 5.6

5.4 替 代 方 案

群集算法存在一类较为简单的实现方法，在该示例中，可创建一个立方体对象，

并将个体对象置于 boid 上。通过 Unity 的刚体物理特性，可简化 boid 的平移和转向行为。为了阻止 boid 间的交叠行为，可添加球体碰撞器组件。

当前实现方案还包含了两个组件，即 boid 个体行为以及控制器行为。其中，控制器表示为其他 boid 所跟随的对象。

Flock.cs 文件中的代码如下所示：

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Flock : MonoBehaviour {
  internal FlockController controller;

  void Update () {
    if (controller) {
      Vector3 relativePos = steer() * Time.deltaTime;

      if (relativePos != Vector3.zero)
          rigidbody.velocity = relativePos;

      //enforce minimum and maximum speeds for the boids
      float speed = rigidbody.velocity.magnitude;
      if (speed > controller.maxVelocity) {
        rigidbody.velocity = rigidbody.velocity.normalized *
            controller.maxVelocity;
      }
      else if (speed < controller.minVelocity) {
        rigidbody.velocity = rigidbody.velocity.normalized *
            controller.minVelocity;
      }
    }
  }
}
```

稍后将创建 FlockController。Update()方法使用 steer()方法并针对 boid 计算速度，并将其用于刚体速度上。随后，将检测刚体组件的当前速度，进而验证是否位于控制器的最大和最小速度限定范围内。若否，则替换现阶段的速度值，对应代码如下所示：

```
  private Vector3 steer () {
    Vector3 center = controller.flockCenter -
        transform.localPosition; // cohesion

    Vector3 velocity = controller.flockVelocity -
        rigidbody.velocity; // alignment

    Vector3 follow = controller.target.localPosition -
        transform.localPosition; // follow leader

    Vector3 separation = Vector3.zero;

    foreach (Flock flock in controller.flockList) {
      if (flock != this) {
        Vector3 relativePos = transform.localPosition -
            flock.transform.localPosition;

        separation += relativePos / (relativePos.sqrMagnitude);
      }
    }

    //randomize
    Vector3 randomize = new Vector3( (Random.value * 2) - 1,
        (Random.value * 2) - 1, (Random.value * 2) - 1);
    randomize.Normalize();

    return (controller.centerWeight * center +
        controller.velocityWeight * velocity +
        controller.separationWeight * separation +
        controller.followWeight * follow +
        controller.randomizeWeight * randomize);
    }
  }
```

steer()方法实现了分离、内聚以及对齐规则，并遵循集群算法中引导者的规则。随后，可利用一个随机权值对上述全部因素予以整合。通过使用包含刚体和球体碰撞

器组件的 Flock 脚本，可生成 Flock 预制组件，如图 5.7 所示。

图 5.7

FlockController 定义了一类简单行为，可在运行期内生成 boid，并更新群集的中心位置和平均速度。

FlockController.cs 文件中的代码如下所示：

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class FlockController : MonoBehaviour {
  public float minVelocity = 1; //Min Velocity
  public float maxVelocity = 8; //Max Flock speed
  public int flockSize = 20; //Number of flocks in the group

  //How far the boids should stick to the center (the more

  //weight stick closer to the center)
  public float centerWeight = 1;

  public float velocityWeight = 1; //Alignment behavior
```

```
//How far each boid should be separated within the flock
public float separationWeight = 1;

//How close each boid should follow to the leader (the more
//weight make the closer follow)
public float followWeight = 1;

//Additional Random Noise
public float randomizeWeight = 1;

public Flock prefab;
public Transform target;

//Center position of the flock in the group
internal Vector3 flockCenter;
internal Vector3 flockVelocity; //Average Velocity

public ArrayList flockList = new ArrayList();

void Start () {
  for (int i = 0; i < flockSize; i++) {
    Flock flock = Instantiate(prefab, transform.position,
        transform.rotation) as Flock;
    flock.transform.parent = transform;
    flock.controller = this;
    flockList.Add(flock);
  }
}
```

代码中声明了全部属性，并以此实现群集算法。随后根据群集输入尺寸生成 boid 对象。另外，代码还定义了控制器类以及父转换类，向 ArrayList 函数中加入了生成的 boid 对象。最后，target 变量接收一个实体，以此作为处于运动状态的引领对象。针对当前群集整体，此处还将创建一个球体实体作为运动过程中的目标引领对象，如下所示：

```
void Update () {
  //Calculate the Center and Velocity of the whole flock group
  Vector3 center = Vector3.zero;
  Vector3 velocity = Vector3.zero;

  foreach (Flock flock in flockList) {
    center += flock.transform.localPosition;
    velocity += flock.rigidbody.velocity;
  }

    flockCenter = center / flockSize;
    flockVelocity = velocity / flockSize;
  }
}
```

Update()方法负责更新平均中心位置以及群集的速度，此类值引自 boid 对象，并用于调整控制器的内聚和对齐属性，如图 5.8 所示。

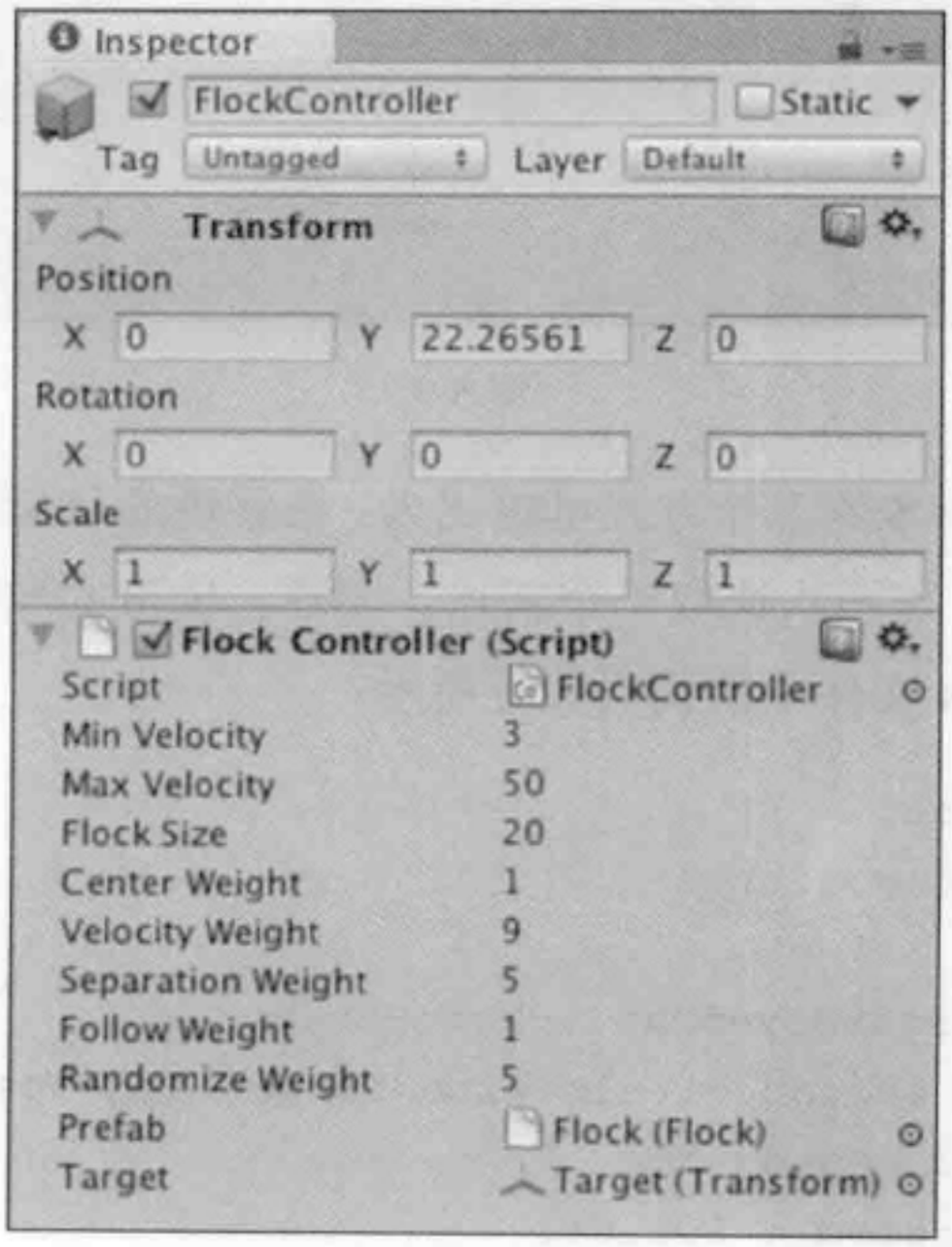

图 5.8

稍后将创建包含 TargetMovement 脚本的 Target 实体，该运动脚本与之前讨论的 Unity 示例控制器运动脚本基本相同，如图 5.9 所示。

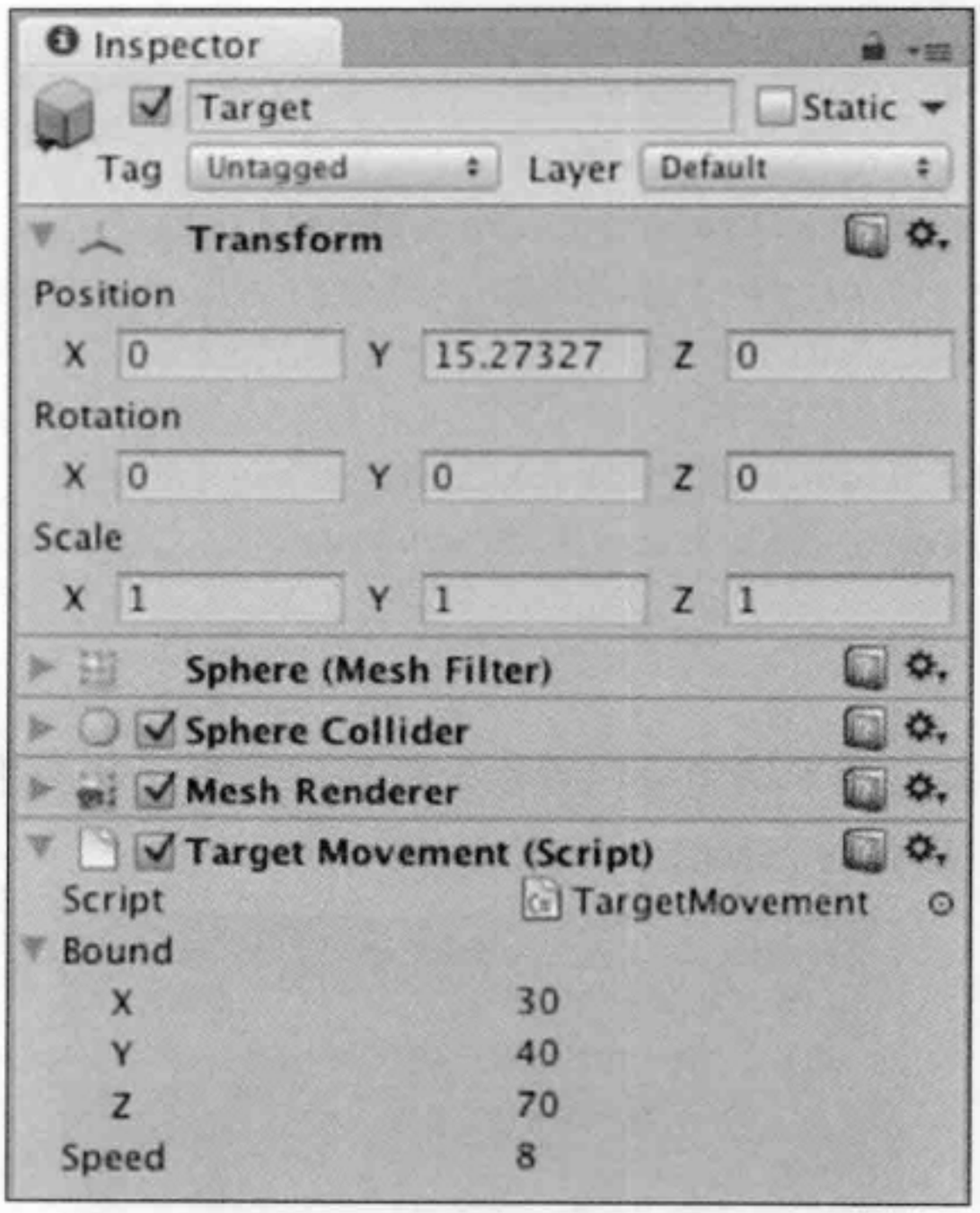

图 5.9

TargetMovement 脚本的工作方式可描述为：在行进的目标位置附近处选取一个随机点，当接近该点时，可再次选取一个新点。随后，boid 对象将到达此目标位置处。

TargetMovement.cs 文件中的代码如下所示：

```
using UnityEngine;
using System.Collections;

public class TargetMovement : MonoBehaviour {
  //Move target around circle with tangential speed
  public Vector3 bound;
  public float speed = 100.0f;
```

```
private Vector3 initialPosition;
private Vector3 nextMovementPoint;

void Start () {
  initialPosition = transform.position;
  CalculateNextMovementPoint();
}
void CalculateNextMovementPoint () {
  float posX = Random.Range(initialPosition.x = bound.x,
     initialPosition.x+bound.x);
  float posY = Random.Range(initialPosition.y = bound.y,
     initialPosition.y+bound.y);
  float posZ = Random.Range(initialPosition.z = bound.z,
     initialPosition.z+bound.z);

  nextMovementPoint = initialPosition+
     new Vector3(posX, posY, posZ);
}
void Update () {
  transform.Translate(Vector3.forward * speed * Time.deltaTime);
  transform.rotation = Quaternion.Slerp(transform.rotation,
     Quaternion.LookRotation(nextMovementPoint -
     transform.position), 1.0f * Time.deltaTime);

     if (Vector3.Distance(nextMovementPoint, transform.position)
        <= 10.0f) CalculateNextMovementPoint();
  }
}
```

待各项数据设置完毕后，群集对象可在当前场景中运动，并朝向目标方向前进，如图 5.10 所示。

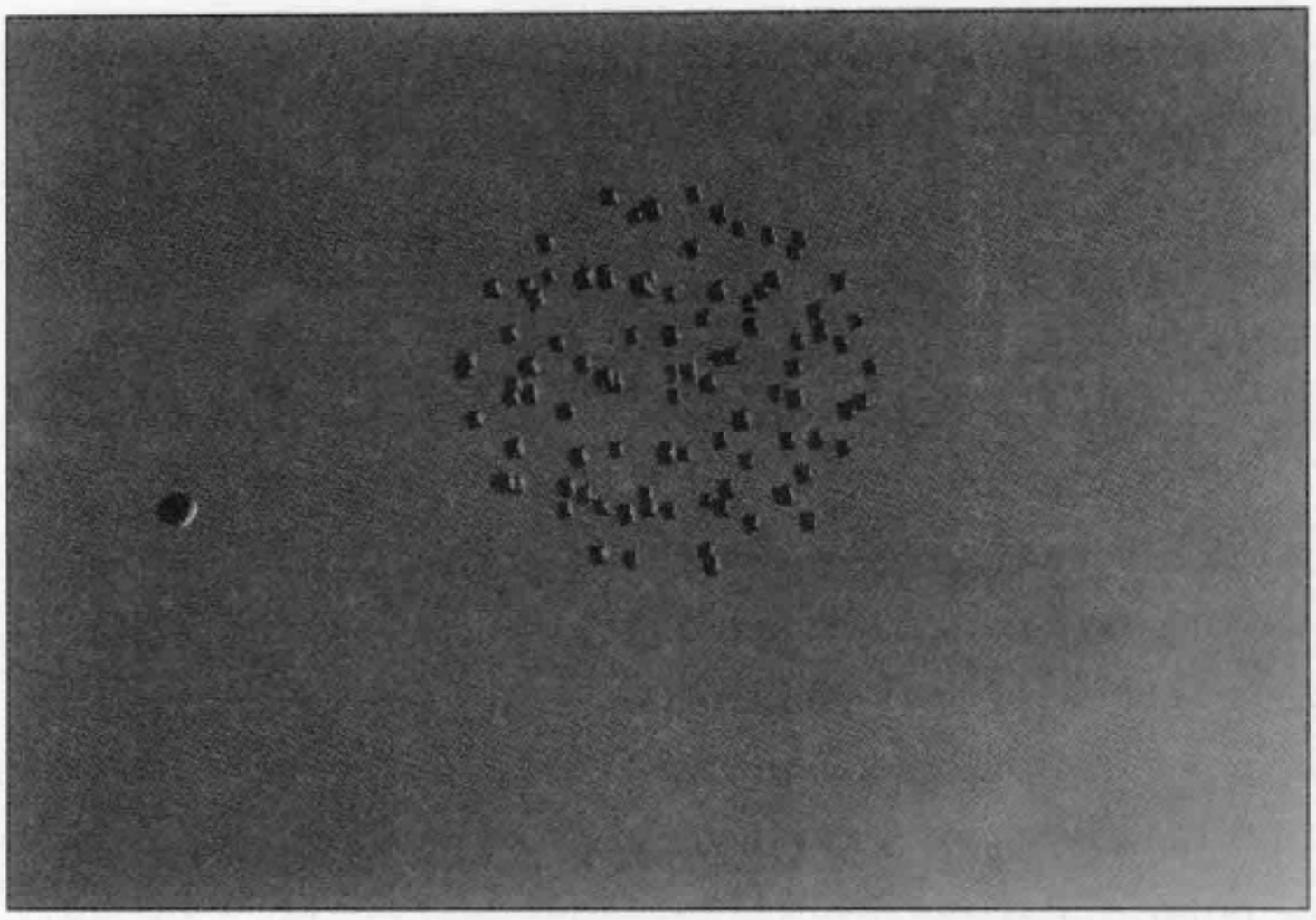

图 5.10

5.5　使用人群群集算法

人群的模拟过程通常较为复杂，实际上并不存在一种有效方法可在通用场景中予以实现。简单地讲，该算法通常是指模拟人类主体对象的群集行为，在某一区域内行进时避免与其他个体和环境产生碰撞。类似于鸟群算法，人群模拟算法广泛地出现于影视业中。例如，电影《指环王》采用了模拟软件 Massive 实现了人群的模拟。这一现象较少出现于游戏业中，尽管如此，某些即时战略游戏中也会出现集结的士兵，并以整体方式穿越场景。

5.5.1　实现简单的群集模拟

当前实现方案相对简单，且主要强调 Unity 中 NavMesh 特性的应用方式。不难发现，NavMesh 承担了许多重大任务。当前场景包含了简单的行进表面，其上包含了烘焙后的 NavMesh。图 5.11 中设置了一组目标对象，以及两组胶囊对象。

通过观察可知，不同对象间彼此相向设置，其设置过程也较为直观：各胶囊对象绑定了 CrowdAgent.cs 组件。当单击播放按钮时，各主体对象朝着目标方向行进，并躲避其他个体以及反方向的胶囊对象。一旦到达目标，主体对象将围绕目标位置而聚集。

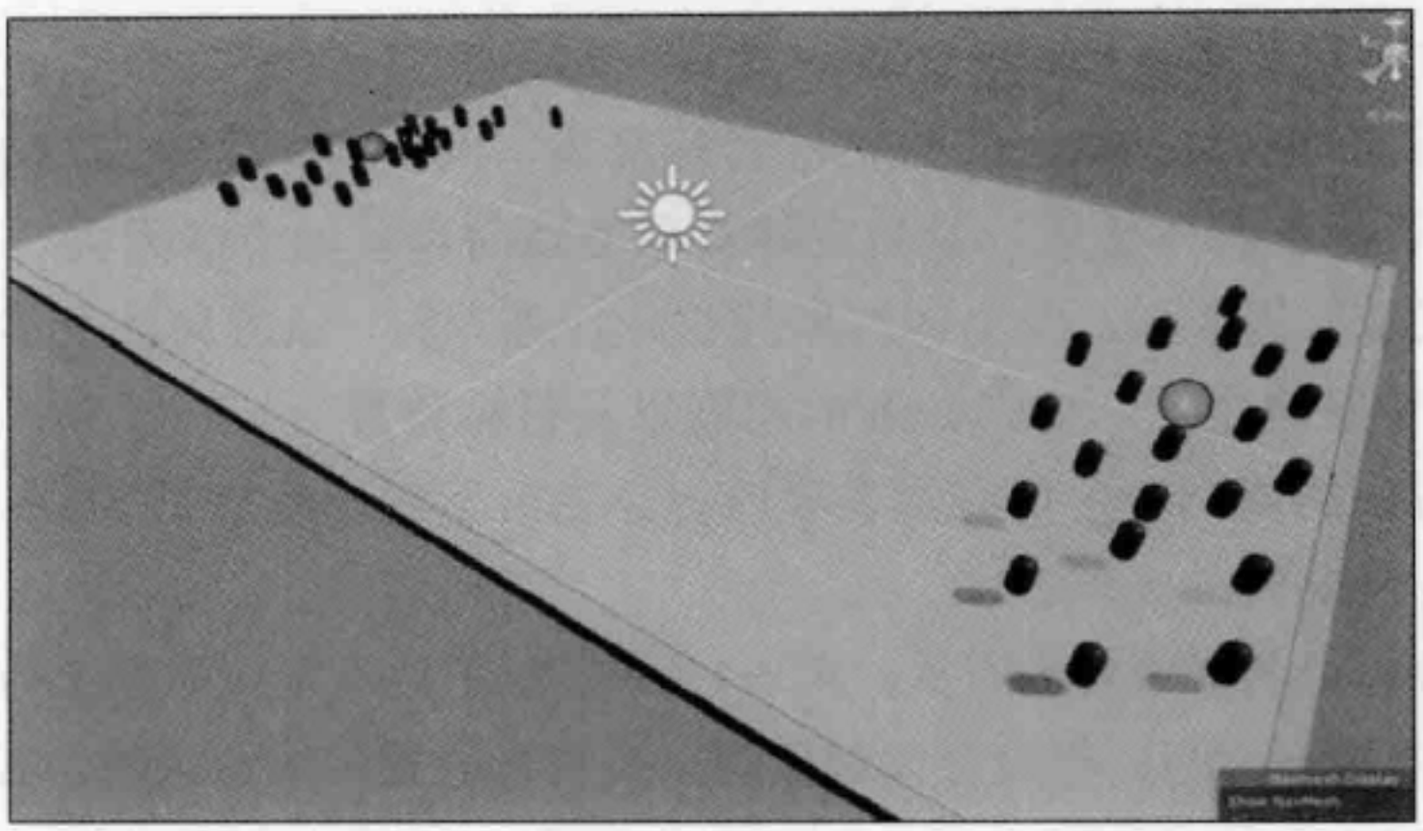

图 5.11

当运行游戏时，可在编辑器中选取单一胶囊对象或一组胶囊对象，并通过可视化方式观察其行为。当导航窗口处于激活状态时，即可查看到与 NavMesh 和主体对象相关的调试信息，如图 5.12 所示。

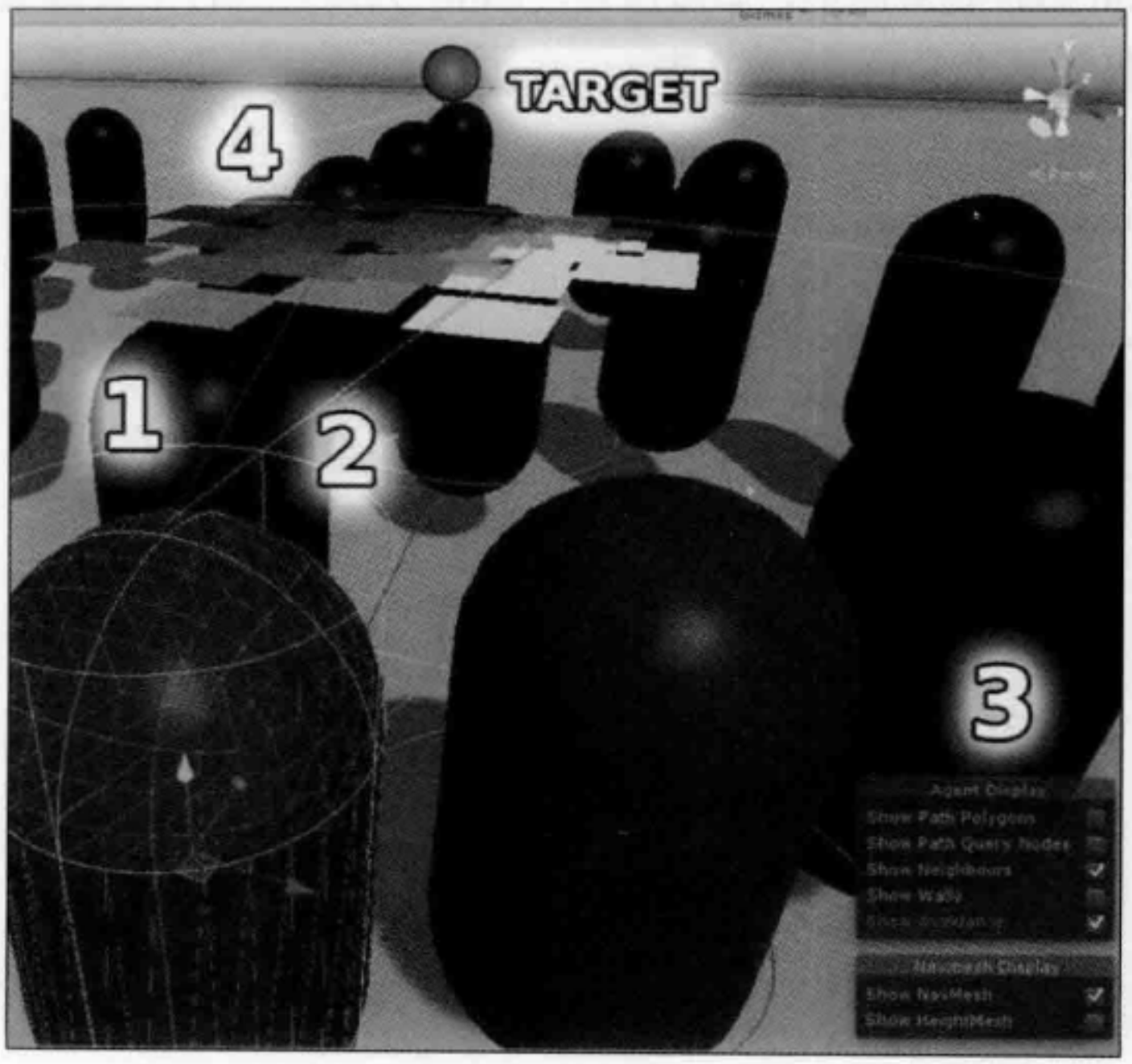

图 5.12

图 5.12 中显示了多个标记，下面对此逐一加以解释。

- 1：指向 NavMeshAgent 目标的目标箭头，对应对象为 RedTarget。此类箭头关注目标的具体位置，且与主体对象的朝向和移动方向无关。
- 2：表示行进箭头，即主体对象的实际行进方向。该方向涉及多个因素，包括邻接对象的位置，NavMesh 的间隔以及目标位置。
- 3：对应的调试菜单可显示多项内容。当前示例中开启了 Show Avoidance 和 Show Neighbours 项。
- 4：该标记与躲避行为有关。图 5.12 中显示了位于主体对象上方、颜色由浅入深的多个正方形，代表主体对象间以及目标位置的躲避区域。其中，深色正方形表示为主体对象密集或者环境阻塞区域；而浅色区域则表示可安全通过的区域。当然，这仅是动态显示结果之一，用户可在编辑器播放过程中查看其变化。

5.5.2　使用 CrowdAgent 组件

CrowdAgent 组件相对简单，如前所述，Unity 为用户执行了大量繁重的工作。下列代码将目标位置赋予 CrowdAgent 中：

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(NavMeshAgent))]
public class CrowdAgent : MonoBehaviour {

    public Transform target;

    private NavMeshAgent agent;

    void Start () {
        agent = GetComponent<NavMeshAgent>();
        agent.speed = Random.Range(4.0f, 5.0f);
        agent.SetDestination(target.position);
    }
}
```

其中，脚本需要使用到 NavMeshAgent 类型的组件，并赋予至 Start()方法中的 agent 变量中。为了进一步提升效果，可在两个值之间随机设置其速度值。最后，可将其目标位置设置为目标标识对象的位置。此处，目标标识通过查看器赋值，如图 5.13 所示。

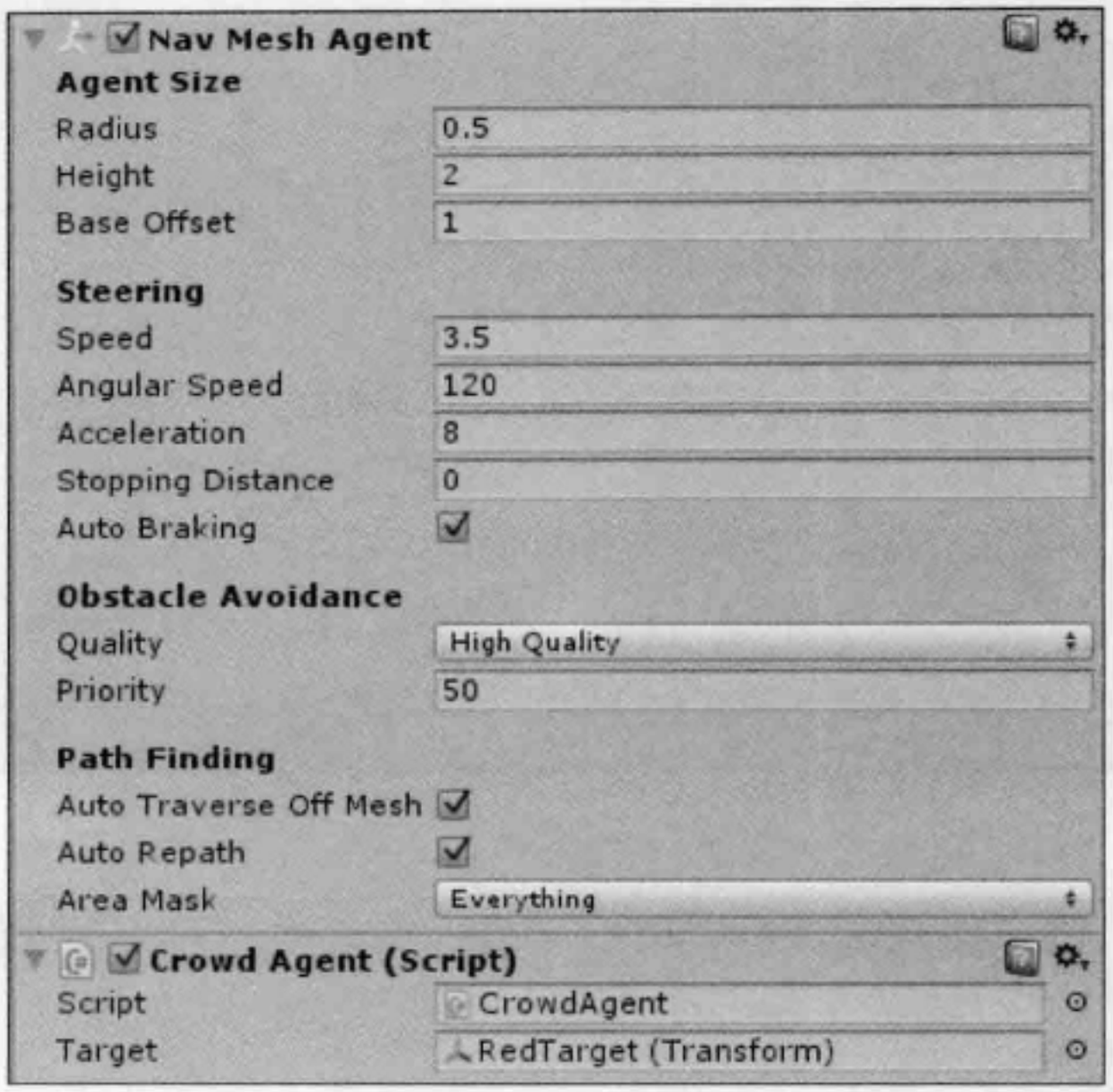

图 5.13

其中，胶囊对象的 Target 项设置为 RedTarget(Transform)。

5.5.3 添加障碍物

对此，可在场景布局中稍作调整，开启 Unity 提供的某些组件，并动态改变主体对象的行为。在 CrowdsObstacles 场景中，可向环境中添加多个墙体，并针对胶囊对象的行进路线创建迷宫结构，如图 5.14 所示。

当前示例最为有趣的部分是各个主体对象的随机速度，其结果在每一时刻均不相同。当主体对象穿越场景时，可能被其他个体会反向行进的主体对象阻塞，进而改变路线或搜索到达目的地的最快路径。这一行为曾在第 4 章中有所讨论，但当前示例包含了大量的主体对象。除此之外，还可向某一墙体以及 NavMeshObstacle 组件添加简单的运动动画（上下运动），如图 5.15 所示。

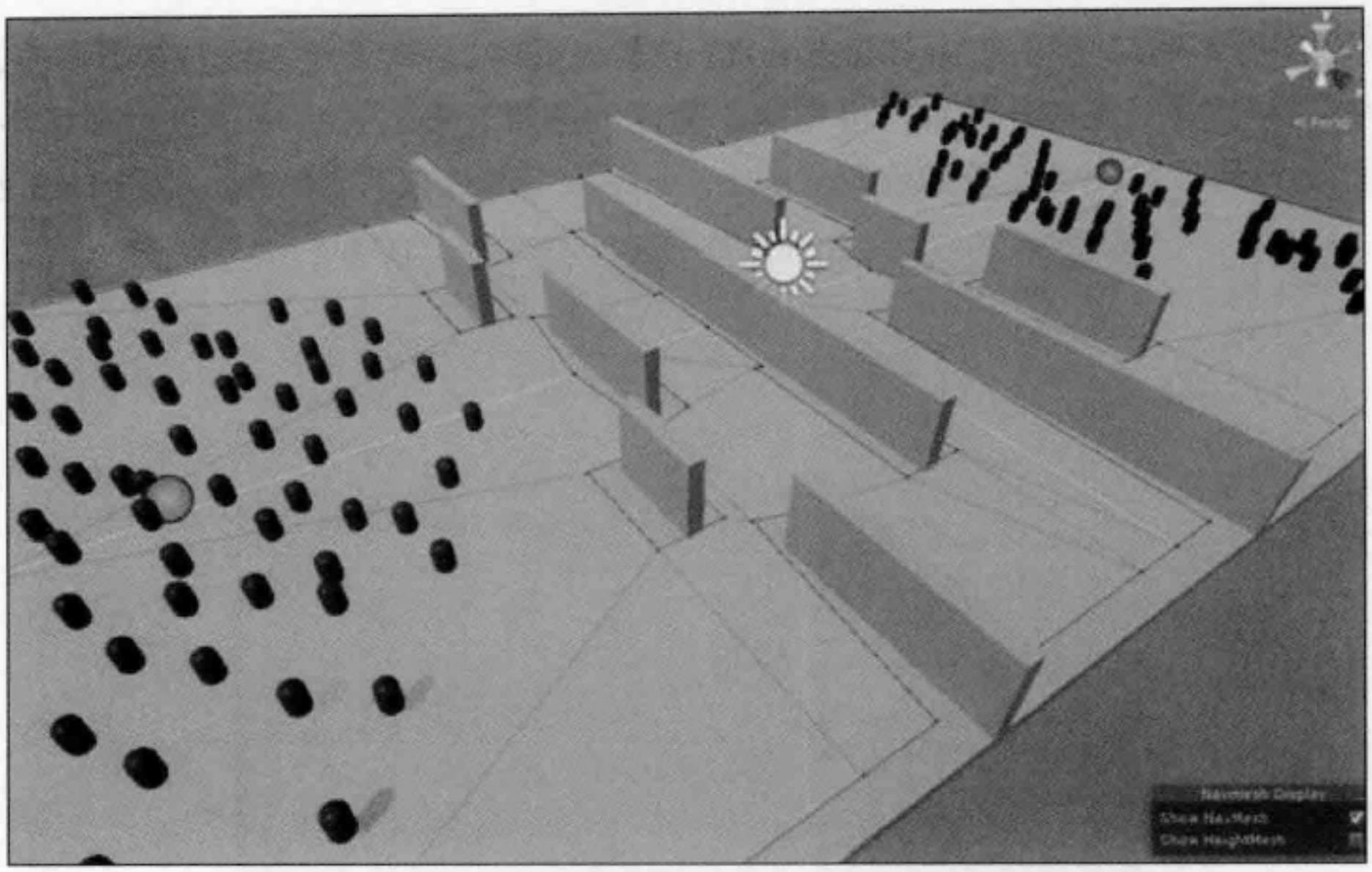

图 5.14

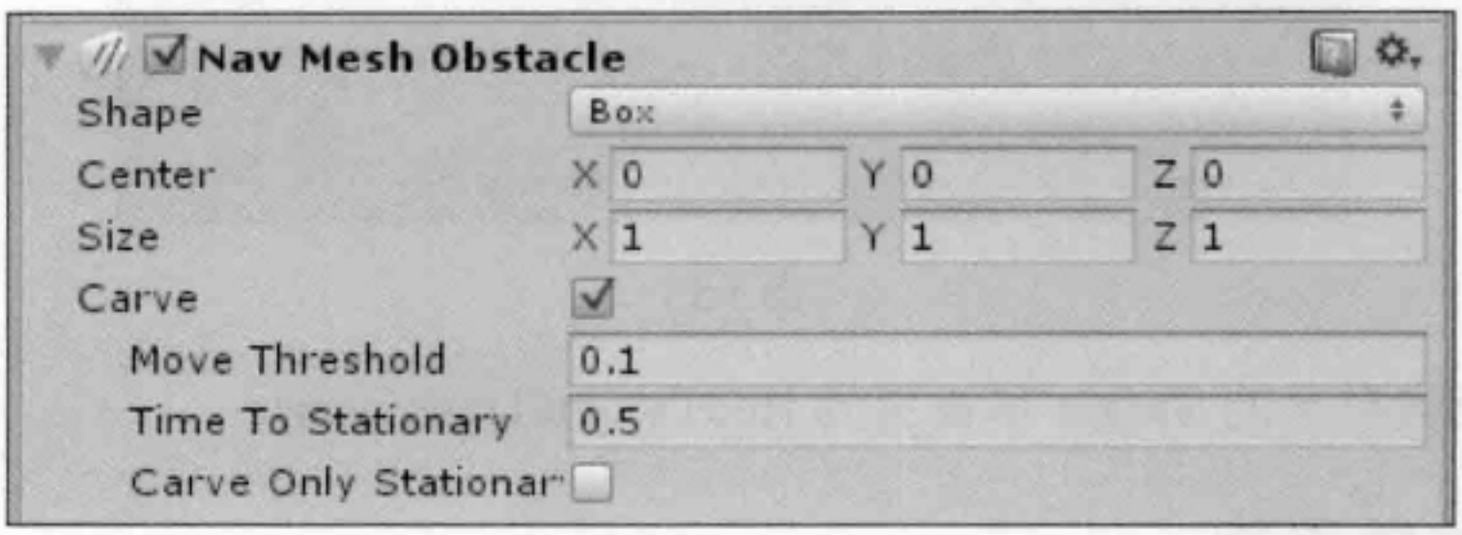

图 5.15

需要注意的是，障碍物在使用过程中不需要设置为 Static。当前障碍物呈现为盒体形状，因而可将默认的 Shape 设置定义为 Box（Capsule 则是另一个选择方案）。Size 和 Center 选项可使用户移动形状的轮廓线并重新调整其尺寸。当前默认设置可与形状实现完美的匹配，因而无须对其进行调整。下一个选项 Carve 则较为重要，并从 NavMesh 中开拓相应空间，如图 5.16 所示。

其中，左图显示了障碍物位于表面上时开拓的空间；在右图中，当障碍物从表面上升起时，NavMesh 将被连接。此处可令 Time to Stationary 和 Move Threshold 保持不变，但应确保 Carve Only Stationary 处于关闭状态，其原因在于，障碍物处于运动状

态，如果未勾选盒体，将会从 NavMesh 中开拓空间。无论障碍物是否处于升降状态，主体角色将穿越障碍物，这并非是当前示例所需要的结果。

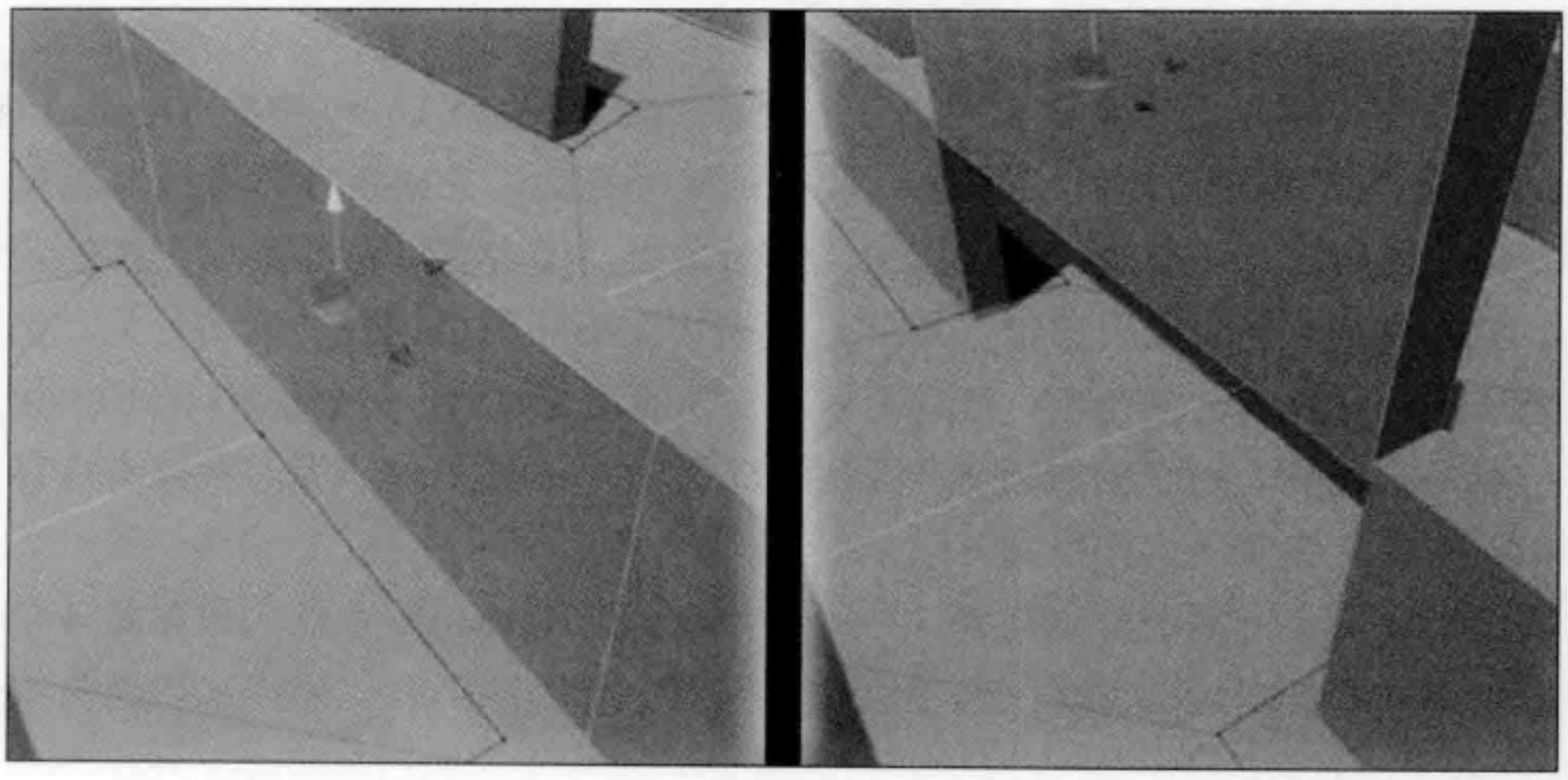

图 5.16

当障碍物处于升降状态，且网格经拓展后重新连接，通过观察可知，主体对象将调整其行进方向。当开启导航调试选项时，可以看到十分有趣的场景，主体对象处于无序行进中，其场面十分混乱，如图 5.17 所示。

图 5.17

5.6　本 章 小 结

本章通过两种方式探讨了群集行为的实现方法，首先是 Unity 中的 Tropical Island Demo 项目；其次则通过刚体对象控制 boid 的运动行为，并采用球体碰撞器避免 boid 间的碰撞。随后，此类群集行为应用于飞行对象上，读者也可将其实现于其他角色行为中，例如鱼群、昆虫以及兽群。对此，仅需实现不同的引领者的运动行为。例如，对于无法实现升降运动的角色，可限定其沿 y 轴的运行行为。另外，2D 游戏中往往会对 y 位置进行限制。对于沿起伏地形上的 2D 运动，还需要进一步调整脚本内容，且不应在 y 方向上施加任何作用力。

最后，本章还考察了群集行为，并通过 Unity 的 NavMesh 系统（参见第 4 章）实现了定制版本的群集运动方案，其中涉及主体对象行为的可视化操作，以及决策处理过程。

第 6 章将讨论行为树模式，并实现定制版本的示例项目。

第6章 行 为 树

行为树（BT）广泛出现于游戏开发中，在最近的几十年里，BT 逐渐成为 AAA 游戏工作室中的选择方案，并可针对主体对象实现相应的 AI 功能。例如，游戏大作《光晕》和《战争机器》均使用了 BT。针对各种类型和领域的游戏 AI，PC、游戏机以及移动设备中强大的计算能力使得行为树成为一种有效的处理方案。

本章主要涉及下列内容：

- 行为树的基本概念。
- 现有行为树方案的优点。
- 如何实现行为树的框架。
- 如何通过框架实现基本的树形结构。

6.1 行为树的基本概念

行为树表示为一类层次结构，并包含了基于公共父节点（根节点）的节点分支系统。行为树模拟了真实世界中的树形结构，这也是其名称的由来。行为树的可视化结果如图 6.1 所示。

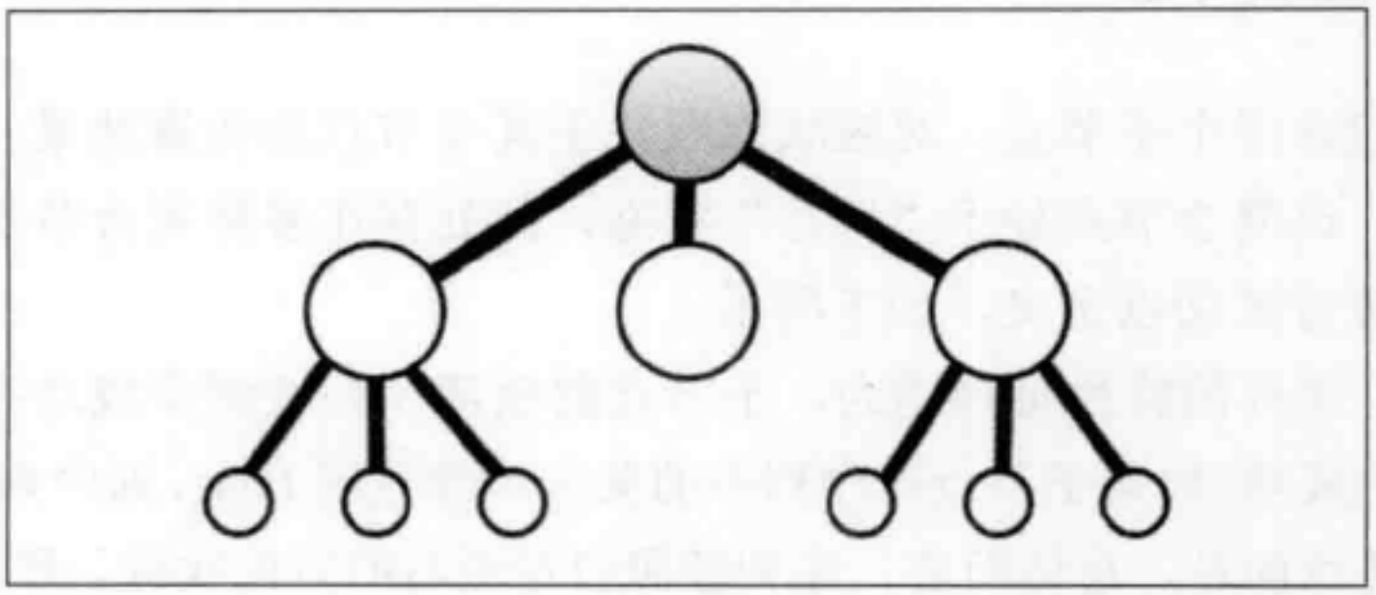

图 6.1

当然，行为树可通过任意数量的节点和子节点构成，其中，层次结构底端的节点称作叶节点，这与真实的树状结构类似。其中，节点可表示为行为或测试，与状态机

不同（依赖于遍历过程中的转换规则），在大型层次结构中，BT 的工作流按照各节点的顺序严格加以定义。BT 首先估算树形结构最上方节点（参见图 6.1），随后持续处理各个子节点，直至满足某种条件或到达叶节点。通常情况下，BT 的估算过程往往始于根节点。

6.1.1　理解不同的节点类型

不同类型的节点名称一般与具体问题相关，甚至节点自身有时也被视为任务。尽管树形结构的复杂度取决于 AI 需求，但如果对各个组件进行逐一考察，则会发现与 BT 工作方式相关的高层概念并不复杂。如果不考虑节点类型，一个节点可返回下列状态之一。

- 成功：节点检测的条件已被满足。
- 无效：节点检测的条件未被满足。
- 运行状态：节点检测的条件的有效性尚未确定，可将其视为“等待”状态。

考虑到 BT 的潜在复杂性，多数实现呈现为异步状态，至少对 Unity 而言情况大致如此。也就是说，估算树形结构时不应阻塞其他操作。在 BT 中，各种节点的估算过程将占用数帧时间。当一次性地估算主体对象上的多个树形结构时，这将对程序性能产生严重的影响，并等待各个节点向根节点返回 true/false，这也是“运行状态”较为重要的原因。

6.1.2　定义复合节点

复合节点包含多个子节点，对应状态取决于其子节点的估算结果。当估算复合节点的子节点时，该复合节点处于“运行”状态，此处存在多种复合节点类型，并通过其子节点的估算方式加以定义，如下所示。

- 序列：序列的特性可描述为，子节点的全部序列按顺序成功完成后，其自身方视为成功。如果子节点在序列中的某一步骤返回 false，则序列自身视为无效。需要注意的是，总体而言，序列按照自左至右的方向执行。图 6.2 和图 6.3 分别显示了成功序列和无效序列。
- 选择器：相比较而言，针对于子节点，选择器则可视为相对宽松的父节点类型。如果选择器序列中的某一子节点返回 true，则选择器将即刻返回 true，

且不再继续估算其他子节点。如果全部子节点估算完毕，且均未返回成功标识，则选择器节点返回 false。

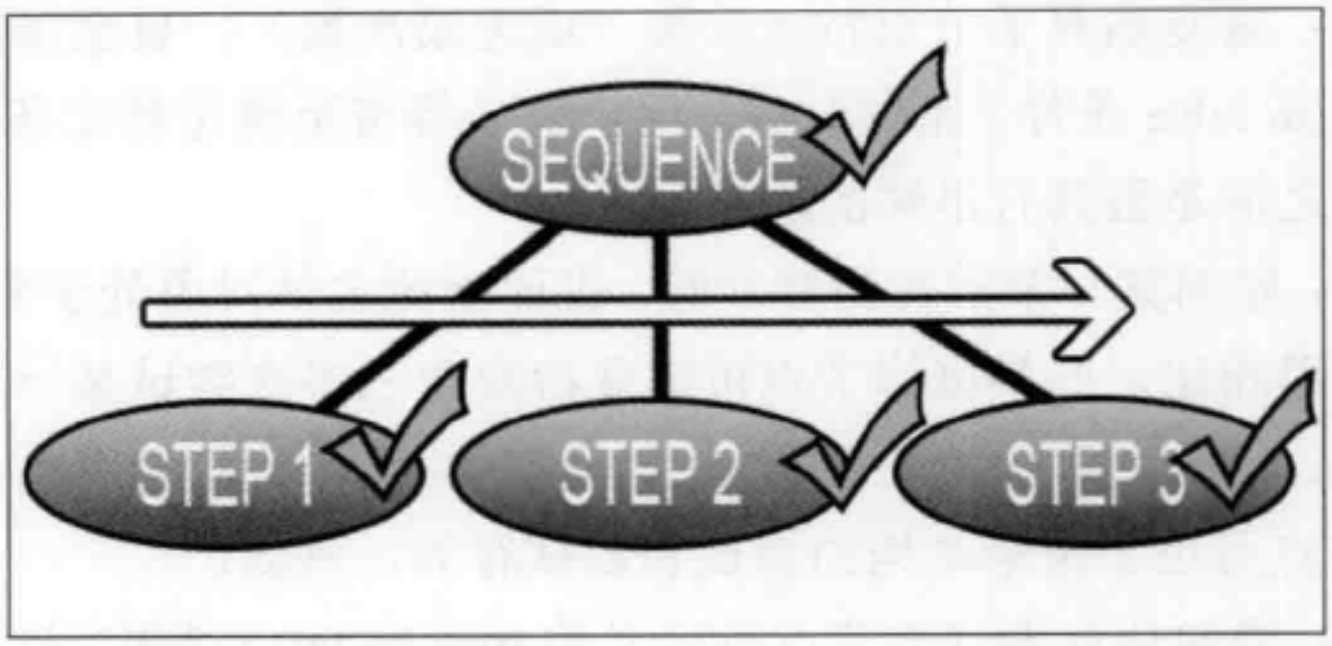

图 6.2

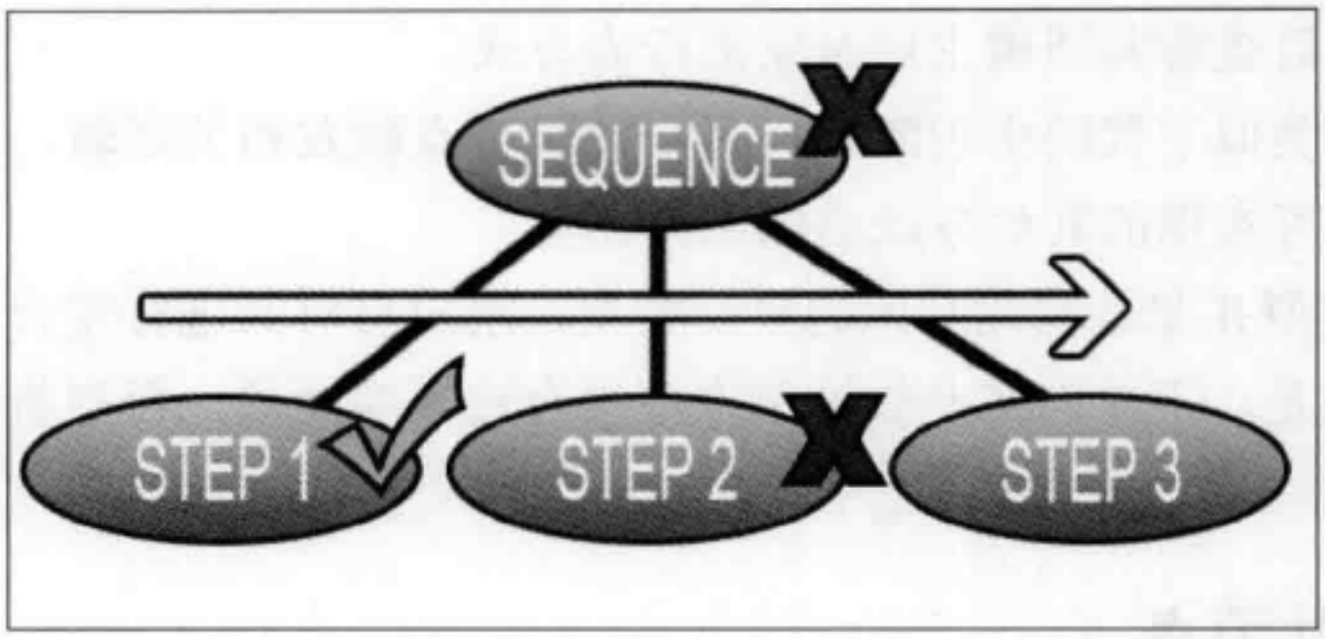

图 6.3

当然，复合节点类型的实际应用视具体情况而定。相应地，可将不同的序列节点类型定义为“与”和“或”条件。

6.1.3 理解修饰节点

复合节点与修饰节点之间最大的差异在于，修饰节点仅包含一个子节点。初看之下，这一行为并无必要，从理论上讲，可在节点自身中包含相关条件，进而获得同样的功能，而非依赖于其子节点。但修饰节点较为特殊，并接收子节点返回的状态，同时根据自身的参数估算响应结果。修饰节点甚至可确定子节点的估算方式和频率。常见的修饰节点类型如下所示。

- 反相器：对此，可将反相器视为 NOT 标识符，并接收与其子节点返回状态相

反的结果。例如，如果子节点返回 true，则修饰节点将其视为 false，反之亦然。这与 C#语言中的！操作符具有相等的功能。

- 重复器：重复估算子节点特定次数（或无限次数），直至满足修饰节点定义的 true 或 false 条件。某些时候，用户需要等待至满足特定条件为止，例如角色攻击之前是否具有足够的能量。
- 限制器：限制某一节点的估算次数，进而避免主体对象处于无限行为循环中。与重复器相比，此类修饰节点可确保游戏角色多次尝试某一动作，例如开启一扇门。

某些修饰节点可用于树形结构的调试和测试行为，例如：

- 假状态：通常估算为修饰节点所定义的 true 或 false 结果。针对于主体对象中断言特定行为十分有效。除此之外，还可令修饰节点维护某一假“运行”状态，进而查看其周围主体对象的行为方式。
- 断点：类似于代码中的断点，可令此类节点触发相关逻辑，并通过调试日志或节点可实现的其他方法通知用户。

上述节点类型并非是彼此互斥的单一类型，用户可对其进行整合，以满足相应需求。需要注意的是，不应向某一修饰节点中组合过多功能项，否则将难以便捷、高效地使用序列节点。

6.1.4　描述叶节点

前述内容曾简要介绍了叶节点以及 BT 结构。在实际操作过程中，叶节点可以是任意行为类型，并可用于描述主体对象所包含的任意逻辑类型。例如，叶节点可定义行走功能、射击命令或踢踏动作。此处不必关注叶节点的操作内容，或者状态的估算方式，读者仅需了解叶节点是层次结构中的最后一个节点，并返回上述 3 种状态之一。

6.2　估算现有方案

Unity 资源商店对于开发者而言是一类不可多得的优秀资源，用户可购买设计素材、音频、插件、框架以及其他各类资源。多数资源均与相关功能密不可分，其中也包含了大量的行为树插件，其价格从免费到数百美元不等。多数资源提供了某种类型

的 GUI，进而实现可视化结果，并对操作体验过程予以简化。

源自资源商店的现有解决方案包含多项优点，大多数框架均提供了高级功能，例如运行期调试机制（基于可视化操作）、机器人 API、序列化机制以及面向数据的树形结构。某些资源甚至包含示例逻辑叶节点，读者可将其应用于游戏中，进而降低代码的编写量。

本书第 1 版中包含了 AngryAnt 发布的 Behave 插件，作为付费插件，当前版本已升至 Behave 2，并可满足开发人员对于行为树的需求。作为一类优秀的框架，Behave 2 兼具健壮性和高效性等特征。

其他方案还包括 Behavior Machine 和 Behavior Designer，其价格和应用特性也不一而同（Behavior Machine 甚至提供了免费版本）。除此之外，读者还可在网络上针对 C#和 Unity 实现获取相关的免费资源。需要指出的是，类似于其他系统，自己亲自编写系统或者使用现有系统，最终方案取决于时间、预算和项目要求。

6.3 实现基本的行为树框架

基于 GUI 的行为树及其多种节点类型的实现过程则超出了本书的讨论范围，本章主要强调核心原理以及相关概念，并针对行为树提供了相对基础的框架。对应示例主要描述简单的操作逻辑，强调树形结构的功能性，而非复杂的游戏逻辑，以使读者明晰游戏 AI 中的概念和工具，进而构建自己的行为树，并对所提供的代码进行有效的扩展。

6.3.1 实现 Node 基类

各个节点需要实现其基本功能，当前框架将包含继承自 Node.cs 抽象基类的全部节点。该类提供了上述基本功能项，或者至少应包含相应的签名，进而对相关功能进行扩展，如下所示：

```
using UnityEngine;
using System.Collections;

[System.Serializable]
public abstract class Node {
```

```
    /* Delegate that returns the state of the node.*/
    public delegate NodeStates NodeReturn();

    /* The current state of the node */
    protected NodeStates m_nodeState;

    public NodeStates nodeState {
        get { return m_nodeState; }
    }

    /* The constructor for the node */
    public Node() {}

    /* Implementing classes use this method to evaluate the desired
set of conditions */
    public abstract NodeStates Evaluate();
}
```

该类较为简单，此处可将 Node.cs 视为其他所构建的节点的“蓝图”。下面首先讨论 NodeReturn 委托类，当前示例并未实现该类。m_nodeState 则表示任意时刻节点的状态，如前所述，对应状态分别为 FAILURE、SUCCESS 或 RUNNING。nodeState 定义为 protected 类型，因而代码其他部分无法直接设置 m_nodeState。

随后是显式定义的空构造函数，虽然该函数并不执行任何操作。最后是 Node.cs 类的核心方法 Evaluate()。在实现了 Node.cs 的相关类中可以看到，该方法用于确定节点的状态。

6.3.2　将节点实现于选取器上

当生成选取器时，可实现前述内容描述的 Node.cs 类中的各项功能，如下所示：

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Selector : Node {
```

```
/** The child nodes for this selector */
protected List<Node> m_nodes = new List<Node>();


/** The constructor requires a list of child nodes to be
 * passed in*/
public Selector(List<Node> nodes) {
    m_nodes = nodes;
}

/* If any of the children reports a success, the selector will
 * immediately report a success upwards. If all children fail,
 * it will report a failure instead.*/
public override NodeStates Evaluate() {
    foreach (Node node in m_nodes) {
        switch (node.Evaluate()) {
            case NodeStates.FAILURE:
                continue;
            case NodeStates.SUCCESS:
                m_nodeState = NodeStates.SUCCESS;
                return m_nodeState;
            case NodeStates.RUNNING:
                m_nodeState = NodeStates.RUNNING;
                return m_nodeState;
            default:
                continue;
        }
    }
    m_nodeState = NodeStates.FAILURE;
    return m_nodeState;
}
}
```

本章前述内容曾有所提及，选取器定义为复合节点，这也意味着，此类节点包含一个或多个子节点，并存储于 m_nodesList<Node>变量中。一种可能是，用户对该类进行扩展，并在该类实例化后加入更多的子节点，但初始状态下，该列表将通过构造

函数予以提供。

代码的下一部分内容展示了前述概念的具体实现过程。其中，Evaluate()方法遍历全部自己算，并分别对其进行估算。鉴于单一无效状态并不意味着选取器整体无效，因而如果某一子节点返回 FAILURE，则可简单地处理下一个子节点。相反，如果任一子节点返回 SUCCESS，则处理过程结束——可设置当前节点的状态并返回该值。如果子节点列表遍历完毕，且不存在任何子节点可返回 SUCCESS，则选取器整体处于无效状态，经赋值后可返回 STATE 状态。

6.3.3　序列的实现

序列的实现方式并无太多新奇之处，Evaluate()方法将采取不同形式对其加以实现，如下所示：

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Sequence : Node {
    /** Chiildren nodes that belong to this sequence */
    private List<Node> m_nodes = new List<Node>();

    /** Must provide an initial set of children nodes to work */
    public Sequence(List<Node> nodes) {
        m_nodes = nodes;
    }

    /* If any child node returns a failure, the entire node fails.
Whence all
     * nodes return a success, the node reports a success. */
    public override NodeStates Evaluate() {
        bool anyChildRunning = false;

        foreach(Node node in m_nodes) {
            switch (node.Evaluate()) {
                case NodeStates.FAILURE:
```

```
            m_nodeState = NodeStates.FAILURE;
            return m_nodeState;
        case NodeStates.SUCCESS:
            continue;
        case NodeStates.RUNNING:
            anyChildRunning = true;
            continue;
        default:
            m_nodeState = NodeStates.SUCCESS;
            return m_nodeState;
                }
            }
            m_nodeState = anyChildRunning ? NodeStates.RUNNING :
    NodeStates.SUCCESS;
            return m_nodeState;
        }
    }
```

基于序列的 Evaluate()方法应针对全部子节点返回 true，在处理过程中，如果某一个子节点返回 false，则全部序列均处于无效状态，这也是首先检测 FAILURE 的原因。相应地，SUCCESS 状态表示随后可继续处理下一个子节点。如果子节点确定为 RUNNING 状态，则输出该节点状态，随后，需要再次估算驱动整体树形结构的父节点或相关逻辑。

6.3.4 将修饰节点实现为反相器

Inverter.cs 对应的结果稍有不同，该类继承自 Node 类，下面将对其实现代码加以考察，并分析其中的不同之处，如下所示：

```
using UnityEngine;
using System.Collections;

public class Inverter : Node {
    /* Child node to evaluate */
    private Node m_node;
```

```
    public Node node {
        get { return m_node; }
    }

    /* The constructor requires the child node that this inverter
decorator
     * wraps*/
    public Inverter(Node node) {
        m_node = node;
    }

/* Reports a success if the child fails and
 * a failure if the child succeeds. Running will report
 * as running */
public override NodeStates Evaluate() {
    switch (m_node.Evaluate()) {
        case NodeStates.FAILURE:
            m_nodeState = NodeStates.SUCCESS;
            return m_nodeState;
        case NodeStates.SUCCESS:
            m_nodeState = NodeStates.FAILURE;
            return m_nodeState;
        case NodeStates.RUNNING:
            m_nodeState = NodeStates.RUNNING;
        return m_nodeState;
        }
        m_nodeState = NodeStates.SUCCESS;
        return m_nodeState;
    }
}
```

不难发现，修饰节点仅包含一个子节点，因而此处采用了单一变量 m_node，而非 List<Node>。相应地，可适当调整代码，并提供空构造函数或某一方法，并在实例化后对子节点进行赋值。

Evalute()方法实现了前述内容所讨论的反相器行为——当子节点为 SUCCESS 时，

反相器返回 SUCCESS；当子节点为 FAILURE 时，反相器返回 SUCCESS；RUNNING 状态则予以正常输出。

6.3.5 创建通用行为节点

ActionNode.cs 类定义为通用叶节点并通过委托方式传递相关逻辑。当然，读者可通过任意方式实现叶节点，并与对应逻辑匹配，只需令该类继承自 Node 类即可。该类兼具灵活性和约束性，其灵活性体现于，可传递任意方法并与委托签名匹配。同时，这也体现了其约束性一面——仅提供单一的委托签名且不接收任何参数，如下所示：

```
using System;
using UnityEngine;
using System.Collections;

public class ActionNode : Node {
    /* Method signature for the action. */
    public delegate NodeStates ActionNodeDelegate();

    /* The delegate that is called to evaluate this node */
    private ActionNodeDelegate m_action;

    /* Because this node contains no logic itself,
     * the logic must be passed in in the form of
     * a delegate. As the signature states, the action
     * needs to return a NodeStates enum */
    public ActionNode(ActionNodeDelegate action) {
        m_action = action;
    }

    /* Evaluates the node using the passed in delegate and
     * reports the resulting state as appropriate */
    public override NodeStates Evaluate() {
        switch (m_action()) {
            case NodeStates.SUCCESS:
                m_nodeState = NodeStates.SUCCESS;
```

```
                return m_nodeState;
            case NodeStates.FAILURE:
                m_nodeState = NodeStates.FAILURE;
                return m_nodeState;
            case NodeStates.RUNNING:
                m_nodeState = NodeStates.RUNNING;
                return m_nodeState;
            default:
                m_nodeState = NodeStates.FAILURE;
                return m_nodeState;
        }
    }
}
```

m_action 委托则是节点正常工作的关键内容。与 C++语言类似，C#语言中的委托机制可视为类型函数指针。另外，也可将委托视为包含（更为准确地讲是指向）某一函数的变量，因而可在运行期内设置被调函数。构造函数需要用户传递与其签名匹配的某一方法，并期望该方法返回 NodeStates 枚举结果。当上述各项条件得以满足后，该方法可实现任意逻辑内容。与之前实现的其他节点不同，当前节点不会产生 switch 语句之外的其他状态，且默认状态为 FAILURE。通过调整默认返回项，用户还可选择 SUCCESS 或 RUNNING 默认状态。

通过继承机制，用户可方便地对该类进行扩展，或者根据需要对其进行调整。除此之外，用户还可忽略此类通用行为节点，并实现特定叶节点的单一版本，但仍需提升代码的复用性。需要注意的是，当前类继承自 Node 类，且应实现所需的相关代码。

6.4　框 架 测 试

框架提供了构建树形结构所需的全部功能项，本节通过人工方式对其予以创建。

6.4.1　行为树的规划

在构建树形结构之前，首先应对实现内容进行适当规划，并在实际操作前实现树形结构的可视化操作。当前树形结构在 0 至特定值之间计数，期间将针对具体值检测

是否符合特定条件，并返回相应的状态结果。树形结构的基本层次结构如图 6.4 所示。

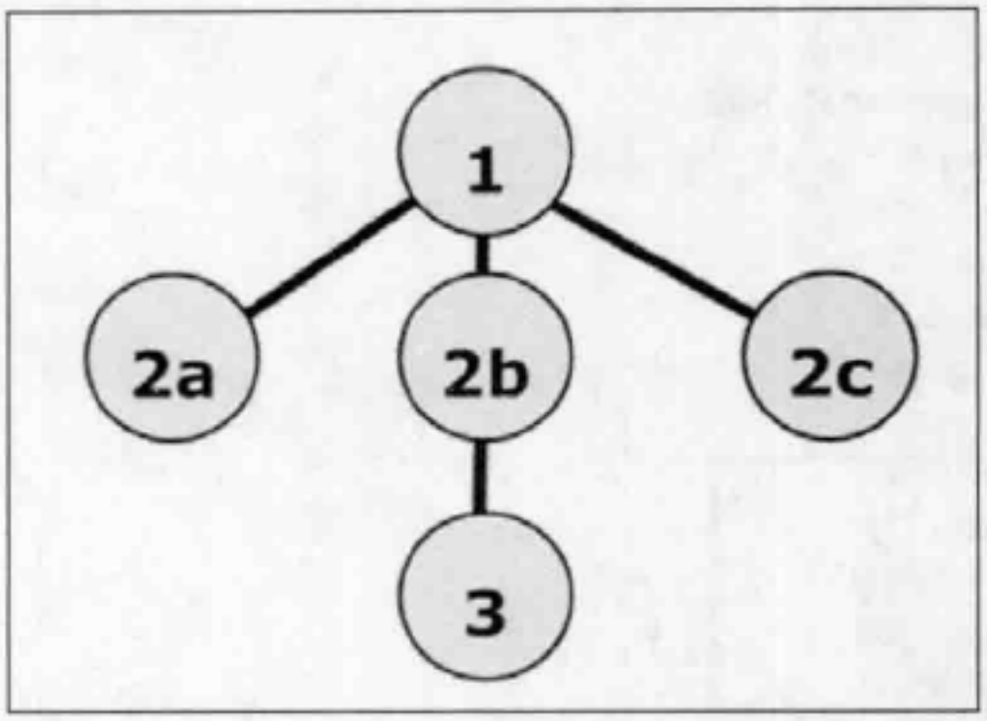

图 6.4

当前测试使用了包含根节点的 3 层树形结构，其中包括以下几个节点。

- 节点 1：该节点表示为根节点，且包含了多个子节点。如果任一子节点返回成功标识，则该节点返回成功标识，因而可将该节点实现为选取器。
- 节点 2a：通过 ActionNode 实现该节点。
- 节点 2b：通过该节点展示反相器的工作方式。
- 节点 2c：根据节点 2a 再次执行相同的 ActionNode，进而查看对树形结构估算结果所产生的影响。
- 节点 3：树形结构中第 3 层中的独立节点，并表示为 2b 修饰节点的子节点。

这意味着，如果该节点返回 SUCCESS，节点 2b 将返回 FAILURE，反之亦然。此时，具体实现细节仍不明晰，但图 6.4 显示了树形结构的可视化结果，在编码时可据此予以实现。

6.4.2 检查场景构建结果

下面考察树形结构的基本结构，在编写具体实现代码之前，此处首先讨论场景的构建方式。图 6.5 显示了树形结构的层次结构，并对相关节点予以标明。

具体构建过程较为简单，即通过正方形简单地显示测试过程中的信息。图 6.5 中所标识的节点将在代码中被引用，并以此对各节点的状态显示相应的可视化结果。图 6.6 显示了当前场景。

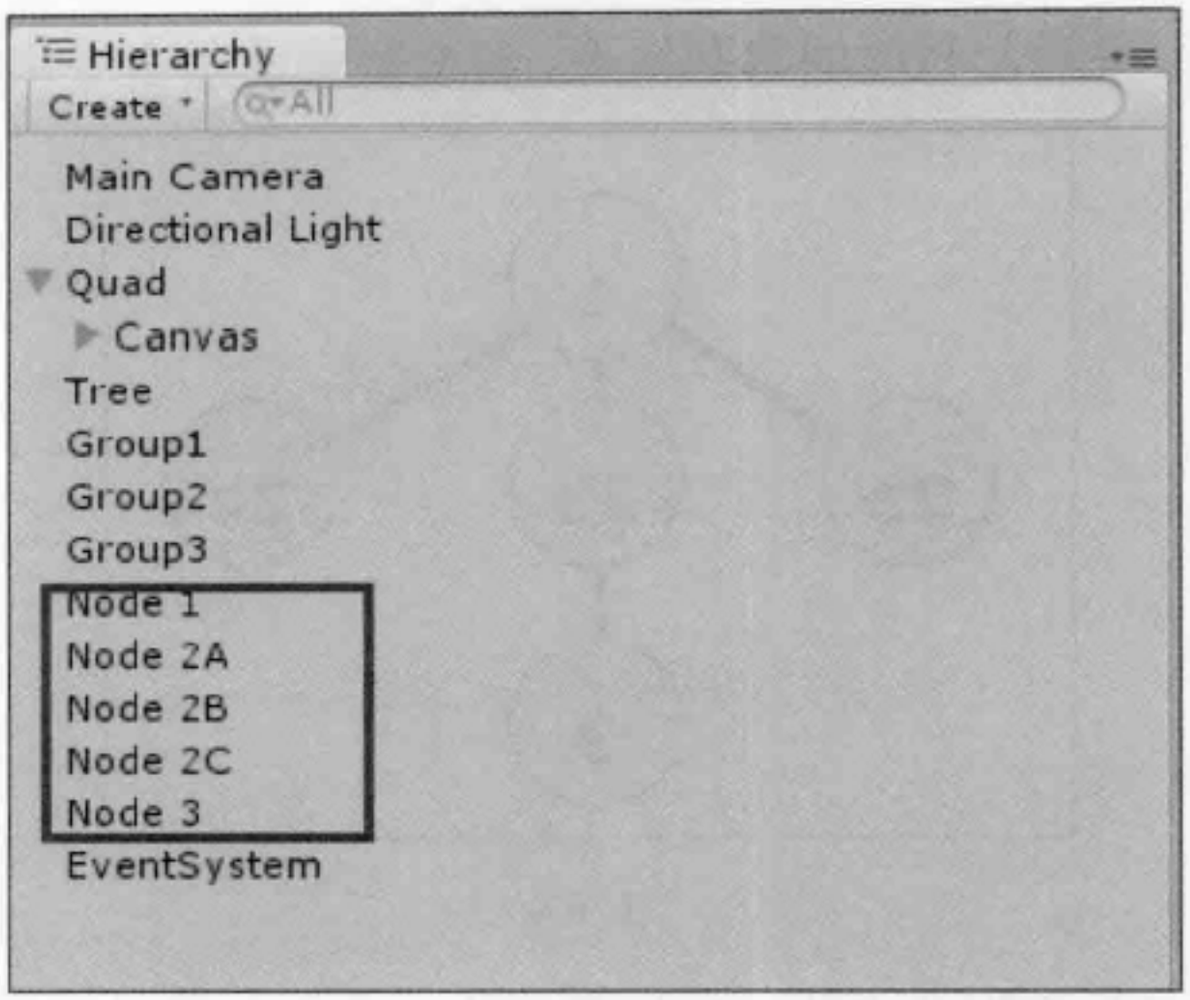

图 6.5

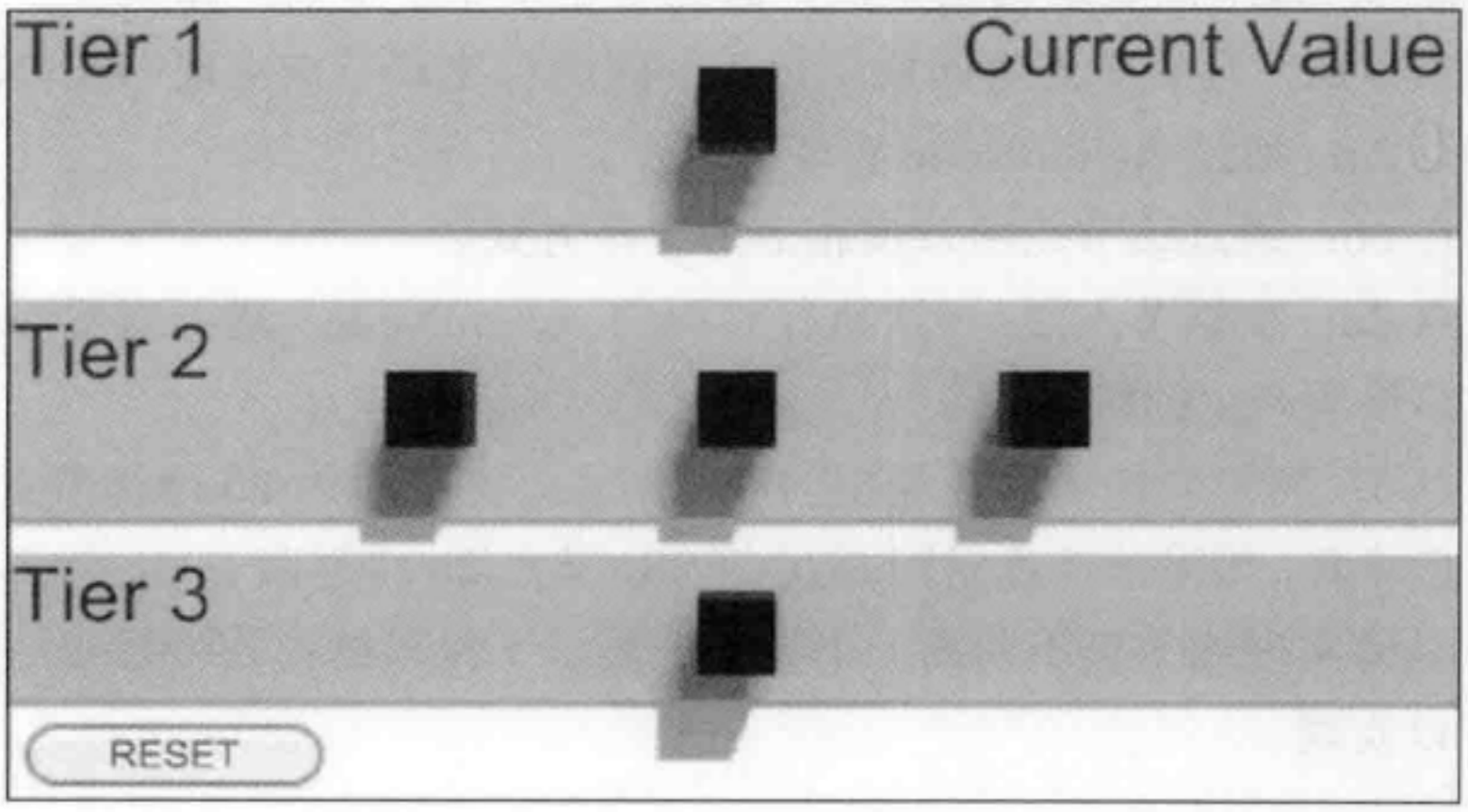

图 6.6

不难发现，此处采用单一节点或盒体表示当前规划过程中所排列的各个节点，并在实际测试代码中被引用，此类节点可根据所返回的状态调整自身的颜色。

6.4.3 考察 MathTree 节点

MathTree.cs 中包含了测试代码，如下所示：

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;
using System.Collections.Generic;

public class MathTree : MonoBehaviour {
    public Color m_evaluating;
    public Color m_succeeded;
    public Color m_failed;

    public Selector m_rootNode;

    public ActionNode m_node2A;
    public Inverter m_node2B;
    public ActionNode m_node2C;
    public ActionNode m_node3;

    public GameObject m_rootNodeBox;
    public GameObject m_node2aBox;
    public GameObject m_node2bBox;
    public GameObject m_node2cBox;
    public GameObject m_node3Box;

    public int m_targetValue = 20;
    private int m_currentValue = 0;

    [SerializeField]
    private Text m_valueLabel;
```

其中，前几个变量用于调试过程；与颜色相关的变量表示为赋予节点盒体的颜色值，以实现其状态的可视化效果。默认条件下，RUNNING 表示为黄色，SUCCESS 表示为绿色，FAILED 表示为红色。

随后将声明实际节点，通过观察可知，m_rootNode 定义为前述内容讨论的选取器。需要注意的是，由于需要向构造函数中传递某些数据，因而此时尚未赋予任何节点变量。

接下来是场景中盒体的引用，即拖曳至查看器中的 GameObjects（稍后将对此加以介绍）。

代码中还定义了一组 int 值，进而可方便地查看代码中的逻辑内容。最后是 UI Text 变量，并可在测试过程中显示相关数据值。

实际节点的初始化过程如下所示：

```
    /* We instantiate our nodes from the bottom up, and assign the
children
    * in that order */
  void Start () {
      /** The deepest-level node is Node 3, which has no children.
*/
      m_node3 = new ActionNode(NotEqualToTarget);
      /** Next up, we create the level 2 nodes. */
      m_node2A = new ActionNode(AddTen);
      /** Node 2B is a selector which has node 3 as a child, so
we'll pass
       * node 3 to the constructor */
      m_node2B = new Inverter(m_node3);
      m_node2C = new ActionNode(AddTen);
      /** Lastly, we have our root node. First, we prepare our list
of children
       * nodes to pass in */
      List<Node> rootChildren = new List<Node>();
      rootChildren.Add(m_node2A);
      rootChildren.Add(m_node2B);
      rootChildren.Add(m_node2C);
      /** Then we create our root node object and pass in the list
    */
      m_rootNode = new Selector(rootChildren);
      m_valueLabel.text = m_currentValue.ToString();
      m_rootNode.Evaluate();
```

```
        UpdateBoxes();
    }
```

出于节点组织方面的考虑，节点将自底向上予以声明，其原因在于：如果未传递子节点数据，则无法实例化父节点，因而需要首先实例化子节点。注意，m_node2A、m_node2C 和 m_node3 表示为行为节点，对此需要传递委托签名。随后，选取器 m_node2B 接收子节点 m_node3。待当前层声明完毕后，由于第 1 层根节点表示为选取器，并需要使用到一个实例化子节点列表，因而应将第 2 层的全部节点置于列表中。

当全部节点实例化完毕后，可通过 Evaluate()方法估算根节点。UpdateBoxes()方法利用相应的颜色值简单地更新之前声明的盒体游戏对象，如下所示：

```
    private void UpdateBoxes() {
        /** Update root node box */
        if (m_rootNode.nodeState == NodeStates.SUCCESS) {
            SetSucceeded(m_rootNodeBox);
        } else if (m_rootNode.nodeState == NodeStates.FAILURE) {
            SetFailed(m_rootNodeBox);
        }

        /** Update 2A node box */
        if (m_node2A.nodeState == NodeStates.SUCCESS) {
            SetSucceeded(m_node2aBox);
        } else if (m_node2A.nodeState == NodeStates.FAILURE) {
            SetFailed(m_node2aBox);
        }

        /** Update 2B node box */
        if (m_node2B.nodeState == NodeStates.SUCCESS) {
            SetSucceeded(m_node2bBox);
        } else if (m_node2B.nodeState == NodeStates.FAILURE) {
            SetFailed(m_node2bBox);
        }

        /** Update 2C node box */
        if (m_node2C.nodeState == NodeStates.SUCCESS) {
```

```
            SetSucceeded(m_node2cBox);
        } else if (m_node2C.nodeState == NodeStates.FAILURE) {
            SetFailed(m_node2cBox);
        }

        /** Update 3 node box */
        if (m_node3.nodeState == NodeStates.SUCCESS) {
            SetSucceeded(m_node3Box);
        } else if (m_node3.nodeState == NodeStates.FAILURE) {
            SetFailed(m_node3Box);
        }
    }
```

由于采用人工方式构建树形结构，因而需要分别对各节点进行检测，获取其 nodeState，并通过 SetSucceeded 和 SetFailed 方法设置颜色。下面讨论该类的核心部分，如下所示：

```
private NodeStates NotEqualToTarget() {
        if (m_currentValue != m_targetValue) {
            return NodeStates.SUCCESS;
        } else {
            return NodeStates.FAILURE;
        }
    }

private NodeStates AddTen() {
        m_currentValue += 10;
        m_valueLabel.text = m_currentValue.ToString();
        if (m_currentValue == m_targetValue) {
            return NodeStates.SUCCESS;
        } else {
            return NodeStates.FAILURE;
        }
    }
```

首先是传递至修饰子节点的 NotEqualToTarget()方法，实际上此处设置了双重否

定条件。如果当前值不等于目标值，该方法返回 SUCCESS，否则返回 FAILURE。随后，父节点反相器计算该节点返回的相反值。因此，如果对应值不等，则反相器返回无效状态，否则返回成功状态。在实际操作过程中，这一点将体现得更为明晰。

下一个方法是 AddTen()，该方法将传递至其他两个行为节点中。顾名思义，该方法向 m_targetValue 变量加 10，并检测是否等于 m_targetValue。若是，则计为 SUCCESS，否则计为 FAILURE。

最后几个方法均具有自解释特征，此处不予赘述。

6.4.4 执行测试

相信读者已经了解了代码的工作理念，下面将对其进行实际操作。首先需要查看相关组件是否以构建完毕。对此，可在层次结构中选择 Tree 游戏对象，对应查看器如图 6.7 所示。

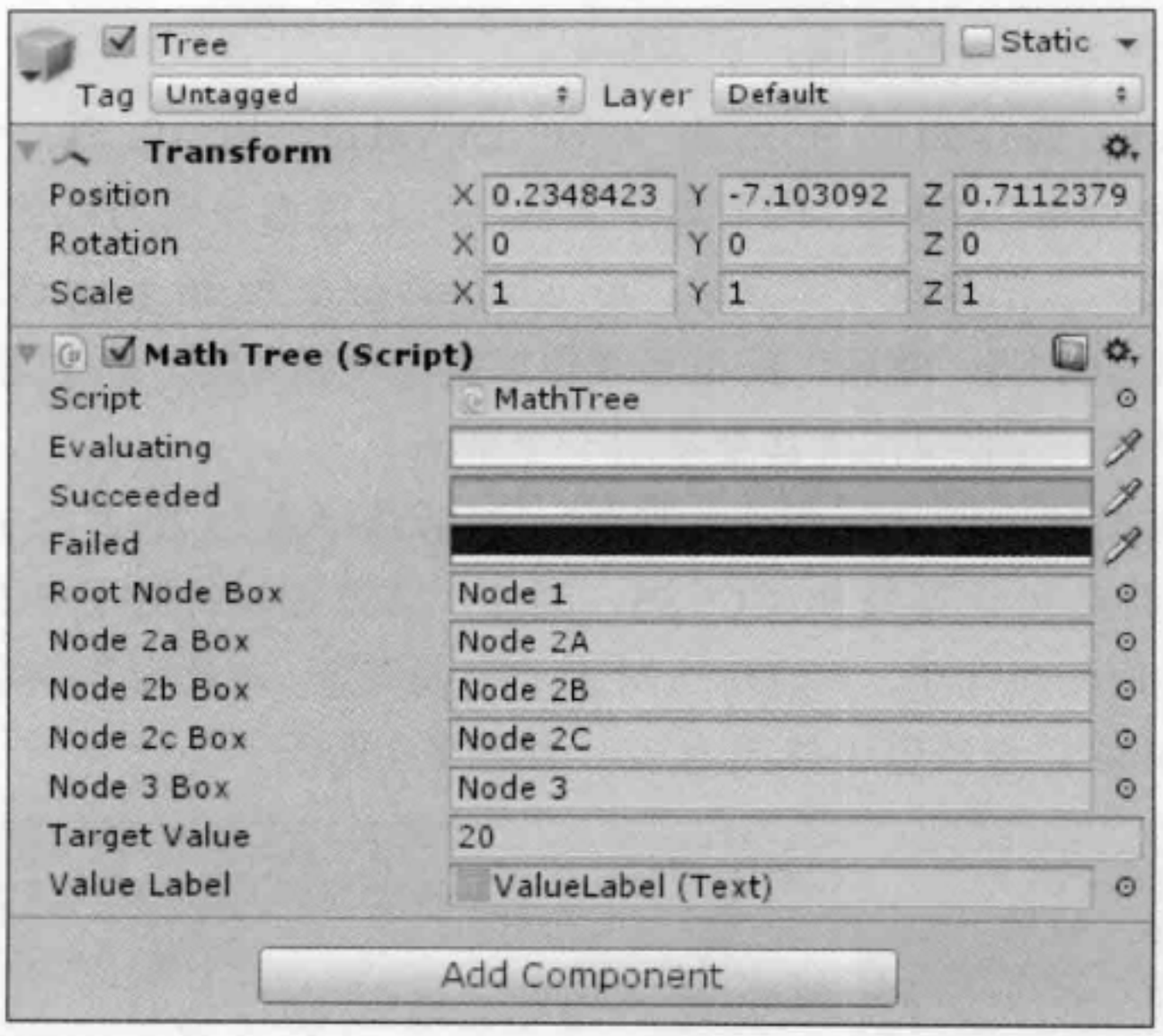

图 6.7

通过观察可知，状态颜色、盒体引用以及 m_valueLabel 均已设置完毕，m_targetValue 遍历也通过代码方式赋值完毕。在单击播放按钮之前，对应值应为 20

（或将其设置为 20）。当播放场景时，盒体将被光源照亮，如图 6.8 所示。

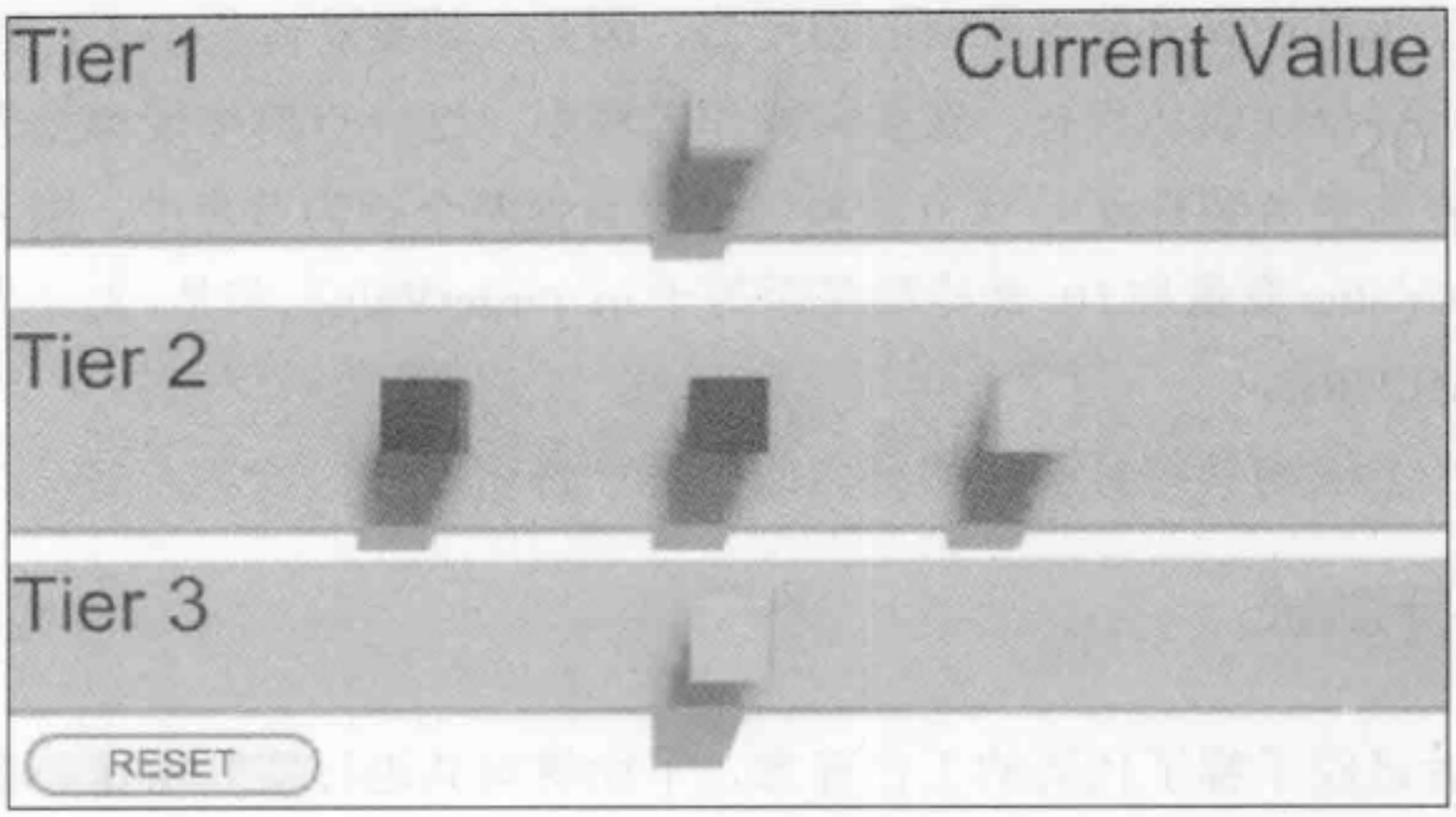

图 6.8

其中，该节点估算为 SUCCESS，这也正是期望的结果，下面将从第 2 层开始逐一进行检查，并解释其中的原因，如下所示。

- 节点 2A：操作始于 0 处的 m_currentValue，加 10 后，该值仍然不等于 m_targetValue（20），因而处于无效状态，进而呈现为红色。
- 节点 2B：当估算其子节点时，m_currentValue 和 m_targetValue 不等，因而返回 SUCCESS。随后，反相器逻辑逆置该响应结果，因而输出 FAILURE 状态。因此，当前操作将移至最后一个节点。
- 节点 2C：向 m_currentValue 加 10，随后结果值为 20 并与 m_targetValue 相等，最终结果计算为 SUCCESS。因此，该节点最终处于 SUCCESS 状态。

上述测试过程相对简单，并清晰地阐述了相关概念。对此可再次运行该测试过程，并将 m_targetValue 在查看器中修改为 30，如图 6.9 所示。

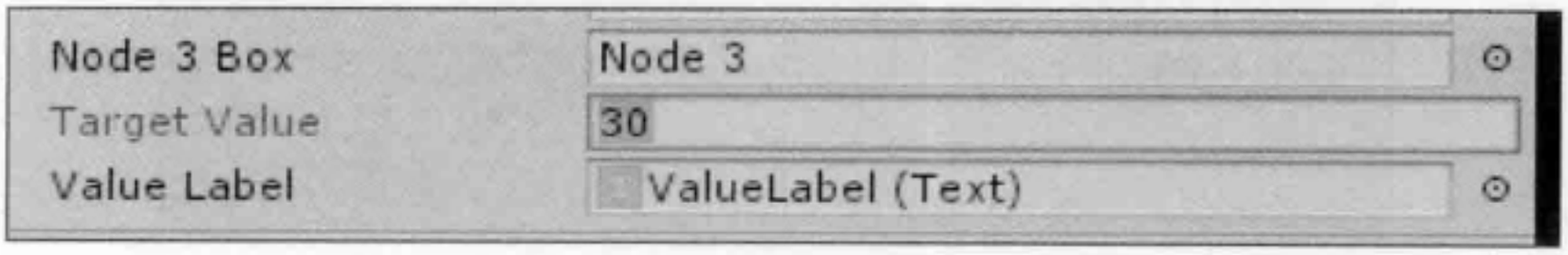

图 6.9

其中的变化并不明显，但会改变树形结构整体间的估算方式。再次播放场景，其中的一组节点将被光源照亮，如图 6.10 所示。

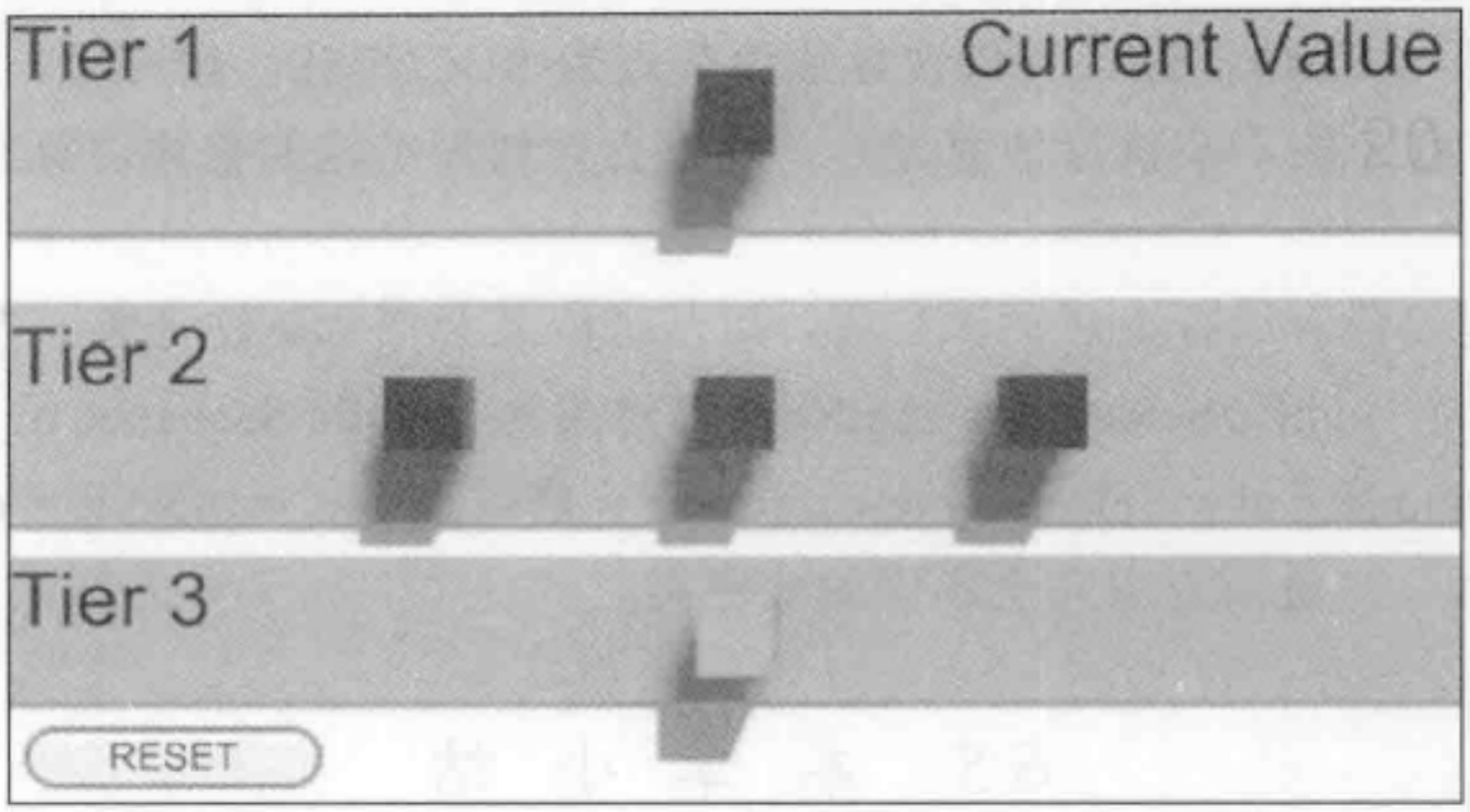

图 6.10

其中，根节点的某一个子节点处于无效状态，因而节点自身输出 FAILURE 信息，下面解释其中的原因。

- 节点 2A：与原示例相比，该节点并无太多变化。m_currentValue 变量始于 0 处，并截止于 10，且不等于 30，因而节点处于无效状态。
- 节点 2B：该节点再次计算其子节点，子节点输出 SUCCESS 信息，因而该节点自身返回 FAILURE，当前操作移至下一个节点。
- 节点 2C：向 m_currentValue 加 10，对应结果累计为 20，变量 m_targetValue 被修改后，将不再返回 SUCCESS 结果。

当前实现中包含了默认值为 SUCCESS 的未估算节点，其原因在于 NodeState.cs 中的枚举顺序，如下所示：

```
public enum NodeStates {
    SUCCESS,
    FAILURE,
    RUNNING,
}
```

在枚举结构中，SUCCESS 定义为第一个枚举值。因此，如果节点未被估算，默认值将不会发生变化。例如，如果希望将变量 m_targetValue 调整为 10，全部节点将呈现为绿色。实际上，这可视为测试实现过程中的附带效果，并非是节点的设计问题。UpdateBoxes()方法将更新全部盒体对象，无论其是否被估算。在当前示例中，节点 2A

经估算后显示为 SUCCESS，并依次导致该节点为 SUCCESS。相应地，节点 2D、节点 2C 以及节点 3 均不在执行估算过程，这不会对树形结构的整体估算结果产生任何影响。

此处，建议读者运行上述测试过程，可从选取器至序列依次调整该节点的实现方式。对此，可将“public Selector m_rootNode;”修改为“public Sequence m_rootNode;”，并将“m_rootNode = new Selector(rootChildren);”修改为“m_rootNode = new Sequence (rootChildren);”，进而查看完全不同的功能集。

6.5 本 章 小 结

本章深入讨论了行为树的工作方式，以及构建行为树的各种节点类型，同时还学习了设置有效节点的不同方案。尽管 Unity 资源商店中提供了大量的现成方案，本章依然采用 C#语言实现了基本的行为树框架，并分析了其中的工作机制。根据现有知识和相关开发工具，本章利用当前框架创建了示例行为树，进而对所涉及的概念进行测试。本章所学知识有助于读者提升游戏行为树的设计水平，并丰富 AI 实现的游戏体验。

第 7 章将通过全新方式向现有概念添加功能项，并提升原有操作的复杂度，还将通过模糊逻辑对行为树和 FSM（参见第 2 章）进行调整。

第7章 模糊逻辑

模糊逻辑通过更为具体的方式表达游戏规则，与其他概念相比，其中涉及了大量的数学知识，大多数信息可通过数学函数表达。对于 Unity 中较为重要的概念，大多数数学内容均得到了不同程度的简化，并通过 Unity 内建特性予以实现。当然，如果读者对数学有着浓厚的兴趣，将会发现此类话题的内容相对艰深。因此，这里建议读者通过相关概念尝试运行相应的示例程序。本章主要涉及以下内容：

- 模糊逻辑的具体内容。
- 模糊逻辑的应用场合。
- 如何实现模糊逻辑控制器。
- 模糊逻辑概念的其他创新性应用。

7.1 定义模糊逻辑

最为简单的模糊逻辑定义方式是与二元逻辑比较后得到的结果，在前述章节中，转换规则可视为 true、false 或者 0、1。相应地，对应结果是否可见？是否位于某一距离之外？即使在计算多个数值时，全部值通常仅包含两个结果，因而可将此类计算过程称作二元逻辑。相比较而言，模糊逻辑涉猎范围更为广泛，其中，各值均被视为浮点数，而非整数。也就是说，此处不再将数值视为 0 或 1，而是将其记为 0~1。

温度则是较为常见的模糊逻辑示例，并可根据不明确数据制定相关决策。例如，漫步在加利福尼亚州夏日的阳光中，即可感觉到气候温暖宜人，.即使游客并不知晓准确的温度值。相反，在阿拉斯加冬日的冰天雪地中，即使不了解当地实际温度值，也能感受到气候的寒冷。其中，寒冷、凉爽、温暖以及炎热均是一类相对模糊的概念。针对冷、暖温差变化，依然存在相应的变化范围可供参考。模糊逻辑可对此类概念建模，并可通过规则集确定问题的有效性和真实性。

当制定相关决策时，事物有时会包含某些灰色地带。也就是说，问题的答案并非是“非黑即白”这一类结果。同一概念也可应用于依赖模糊逻辑的主体对象上，例如，如果角色感到饥饿，是否可确定对应的临界点？对此，可将餐后时刻定义为 0，而 1

则表示为饥饿点，如图 7.1 所示。

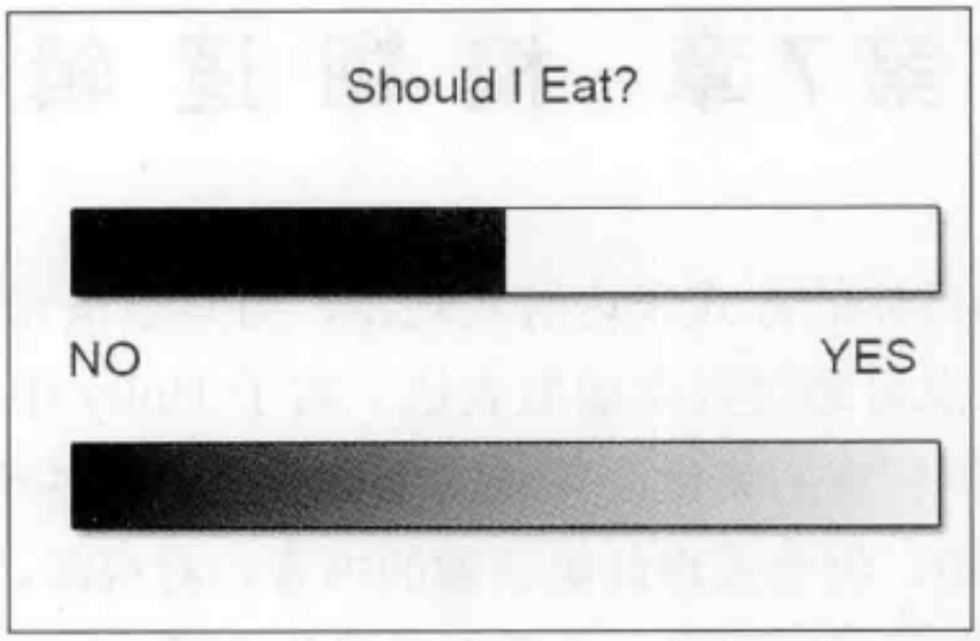

图 7.1

当制定对应决策时，存在多种因素可对选择结果产生影响，这将引出模糊逻辑控制器这一问题，进而对大量数据予以考察。在上述示例中，仅考察了与决策相关的单一值，即距上次进餐的时间值。然而，还存在其他因素可对当前决策的制定产生影响，例如能量消耗，以及运动量。无论如何，读者可以看到存在多个输入值可对输出结果产生影响，且均可作为进餐的条件。

模糊逻辑系统根据其通用特性可体现得异常灵活。期间，用户提供输入数据，模糊逻辑系统将提供输出结果，该结果的具体内容完全取决于用户自身。此处主要考察输入数据对象决策的影响方式，获取输出结果，并以计算机和主体角色可理解的方式对其加以应用。然而，输出结果还可用于确定执行的任务量、频率以及时长。例如，假设主体对象为赛车类游戏中的车辆，并包含 Nitro 快速启动装置。其中，0 或 1 表示启动的常规时间量，或者供使用的正常燃料量。

类似于本书之前所讨论的系统，以及游戏程序设计中的大多数理念，当制定问题处理的最佳方案时，依然需要估算游戏的各项需求条件以及硬件的局限性。

可以想象，简单的 yes/no 系统至复杂的模糊逻辑系统间均存在性能开销，这也是选取应用方式的原因之一。当然，复杂系统并不意味着最优方案，某些时候，二元系统的简单性、可预测性可能更加适合。

相信读者均对“简单即是美”这一谚语有所耳闻，但凡是不可过于简单。相对论之父爱因斯坦常引用的一句话则是“简约不简单”。相应地，角色 AI 也应满足游戏的要求，但不可过于简陋。例如，游戏 Pac-Man 中的 AI 即可与其实现良好的系统工作，整体 AI 系统较为简洁。然而，这一规则可能并不适用于现代射击游戏或即时战

略游戏。

通过本书所提供的知识和相关示例，相信读者可获取最佳解决方案。

7.2 模糊逻辑应用

一旦读者理解了模糊逻辑背后隐藏的简单概念，即可考察其多种应用方式。在实际操作过程中，不同的工作需要用到不同的工具。

模糊逻辑系统将使用到某些数据，其计算方式与人类获取信息的方式类似，并于随后将数据转换为系统可用的信息（对于人类而言，这一过程相对简单）。

在现实生活中，模糊逻辑控制器包含多项应用。尽管游戏 AI 中无法与此一一对应，但总结起来模糊逻辑控制器包含以下几点内容。

- 采暖通风和空调（HAVC）系统：关于模糊逻辑，前述温度示例不仅显示了较好的理论方案，同时也是真实世界中较为常见的模糊逻辑控制器示例。
- 车载系统：现代汽车均配备了高级的计算机系统，包括空调系统，燃料供给以及自动制动系统。实际上，与早期二元系统相比，车载计算机则是一类更为高效的系统。
- 智能手机：手机的明暗程度常与环境光紧密相关。现代智能手机操作系统会考虑环境光、显示数据的色彩效果，以及当前电池容量，进而对屏幕的亮度进行优化。
- 洗衣机：大多数现代型洗衣机（最近 20 年间的产品）均不同程度地采用了模糊逻辑，包括载荷、水流的清洁程度、温度以及循环过程中的其他因素，进而对用水、能耗以及时间进行优化。

如果读者此时对房间环境加以观察，将会发现诸多与模糊逻辑相关的事物（当然，此处并不包含计算机设备）。尽管这体现了上述概念的实际应用结果，但人们已对此司空见惯，或者说，这一类应用并未实际应用于游戏中。鉴于本人对于游戏有着浓厚的兴趣，其中包括精灵巫师、魔法以及怪兽，下面将对相关示例加以考察。

7.2.1 实现简单的模糊逻辑系统

当前示例设定了精灵巫师 Bob 这一角色，Bob 生存于 RPG 游戏世界中，并拥有

强大的治愈魔法。Bob 需要根据自身的生命值（HP）确定何时施展魔法。

在二元系统中，Bob 的决策系统如下所示：

```
if(healthPoints <= 50)
{
  CastHealingSpell(me);
}
```

不难发现，Bob 的生命值处于两种状态下——大于 50 或小于等于 50，该过程并无任何不当之处。该方案的模糊逻辑版本也有几分类似之处，首先需要确定 Bob 的生命值状态，如图 7.2 所示。

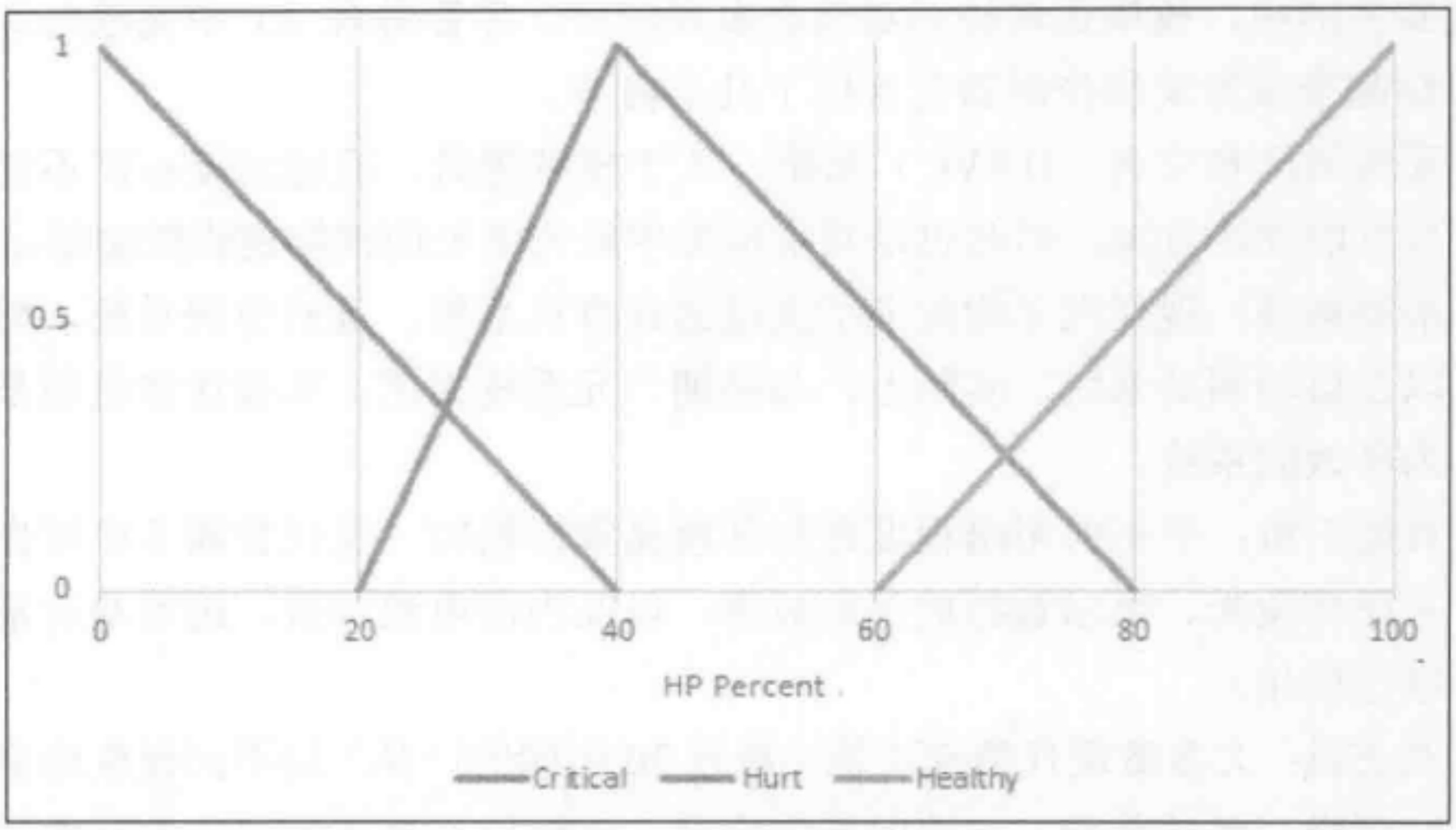

图 7.2

其中，第一条线段将 Bob 施展魔法的概率映射至所减少的生命值上。简而言之，该过程表示为一个线性函数，且与模糊逻辑并无太大关系——这仅体现了一种线性关系，作为复杂二元系统中的一个步骤，其中缺少了“模糊”元素。

下面考察隶属函数这一概念，且对当前系统至关重要，进而可确定某一逻辑的正确程度。在当前示例中，并非仅是简单地对原始数据进行考察，进而确定 Bob 施加魔法的时刻。相反，此处可针对 Bob 将其分解为多个逻辑信息块，随后判断 Bob 的行为过程。

下面查看并比较 3 个逻辑问题，并对结果进行估算。其中不仅涉及各个问题的正确程度，还需进一步从中选取最佳结果，如下所示：

- Bob 伤情较为严重。
- Bob 处于受伤状态。
- Bob 处于健康状态。

从正式术语来看，可将隶属程度定义于某一集合中，随后，主体对象可据此确定下一步执行的任务。

不难发现，两个逻辑于某一时刻可同时为 true。例如，Bob 受伤并处于临界状态下。除此之外，Bob 仅受到轻微伤害，且生命值稍有损失。对此，可分别设置相应的阈值，下面根据图 7.2 估算状态。其中，作为规范化的浮点值（0~1），垂直数值表示逻辑的正确程度，具体如下：

- 在 0%生命值处，可以看到临界逻辑值计算为 1。该结果表达了如下事实：当失去全部生命值后，Bob 将处于临界状态。
- 在 40%生命值处，Bob 处于受伤状态，这也是当前最真逻辑结果。
- 在 100%生命值处，Bob 处于健康状态，这可视为当前的真实结果。

位于真逻辑之外的状态则处于逻辑边界中，例如，Bob 的生命值位于 65%处，对应结果如图 7.3 所示。

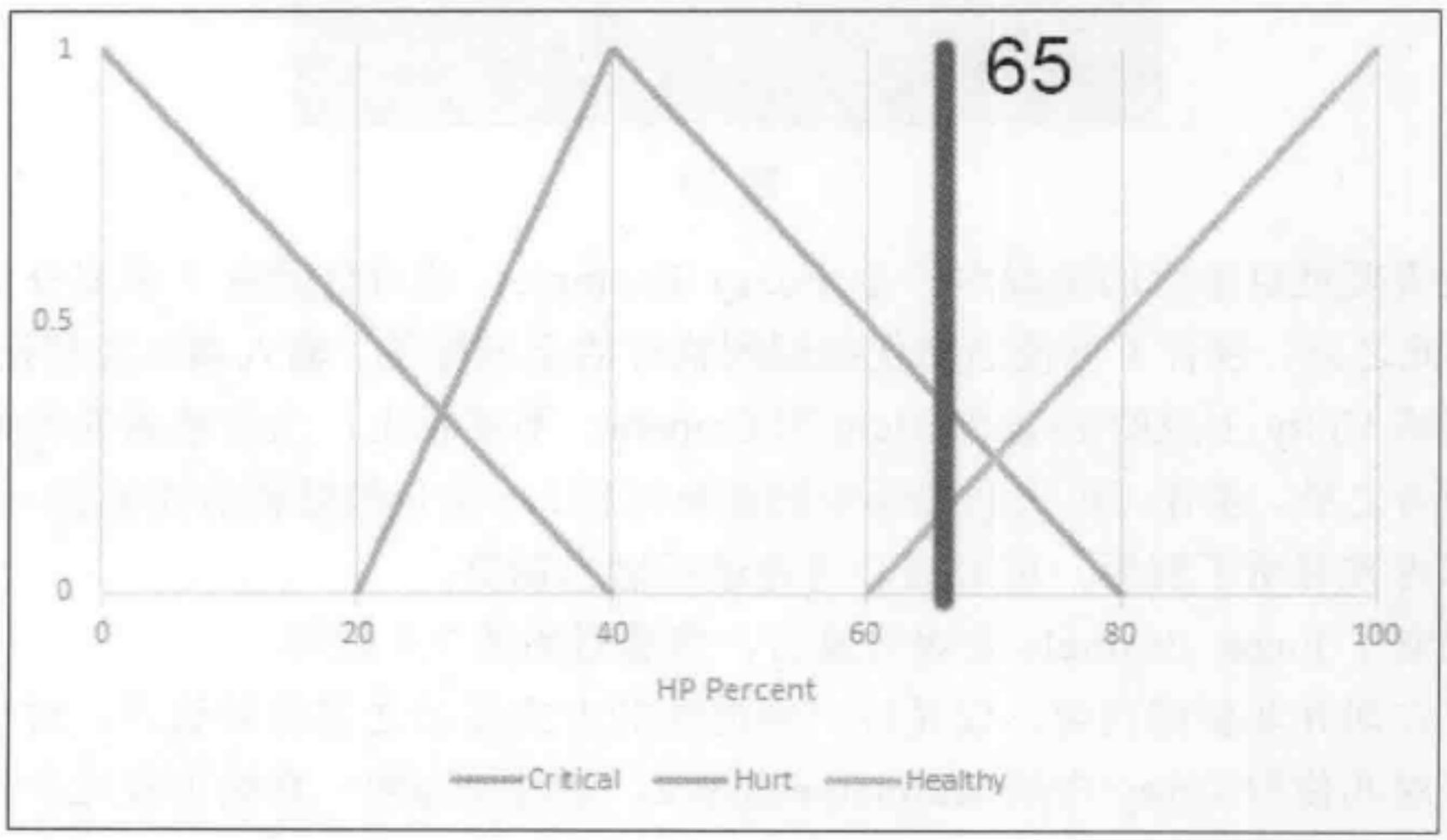

图 7.3

其中，对应位置处绘制了一条垂直线，并与两个集合分别相交。这也意味着，Bob 身负轻伤，但仍然处于某种健康状态。通过观察可知，垂直线在上方点处与“受伤状

态”集相交。据此可知，Bob 处于相对严重的受损状态。特别地，Bob 的生命值受到 37.5%的伤害，健康程度为 12.5%，临界状态为 0%。下面通过代码形式考查这一问题，打开 Unity 中的 FuzzySample 示例场景，对应层次结构如图 7.4 所示。

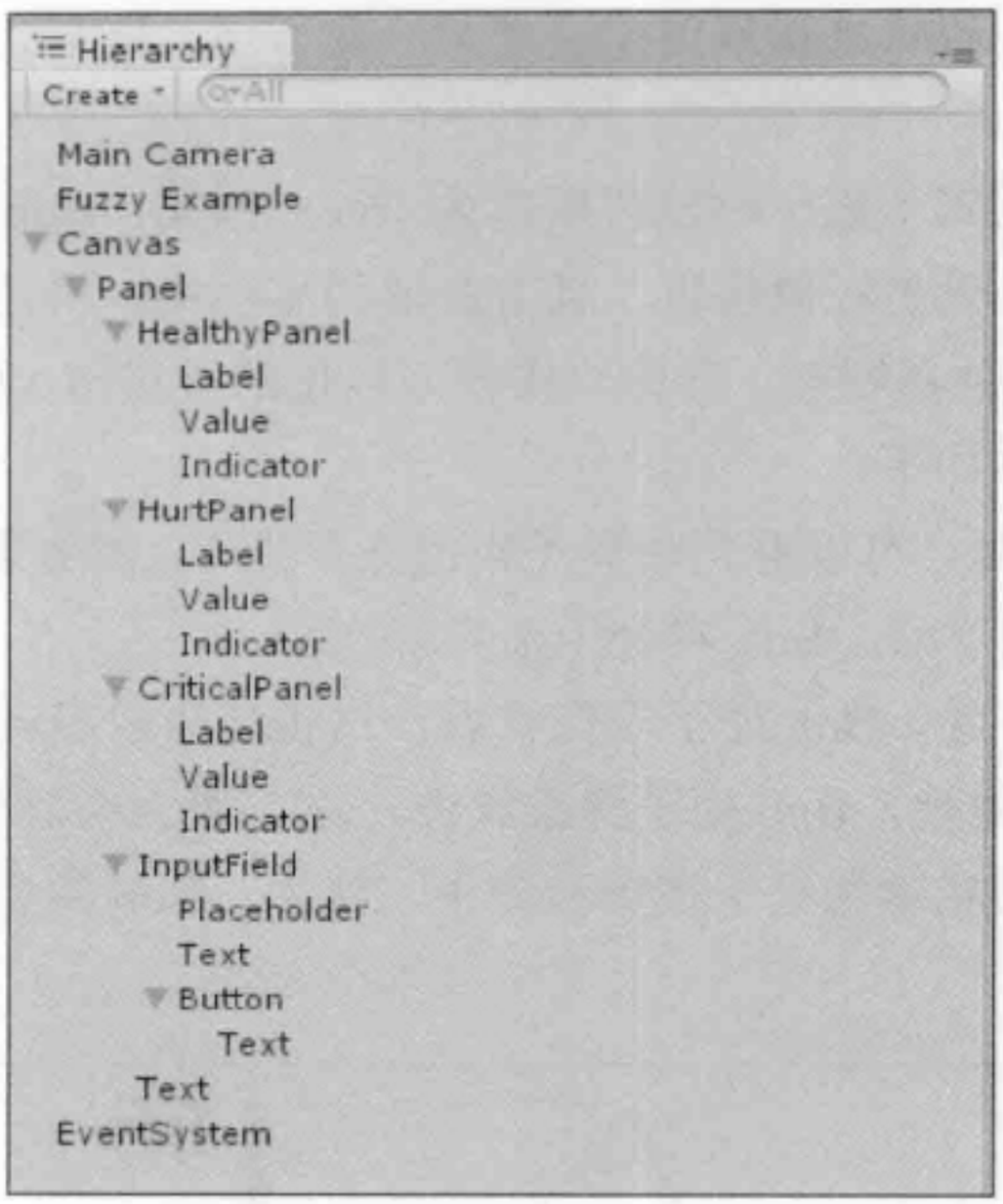

图 7.4

此处需要重点考察的游戏对象是 Fuzzy Example，该对象涵盖了需要分析的相关逻辑。除此之外，包含了可使当前任务顺利执行的全部标签、输入项以及按钮。最后，此处可忽略 Unity 生成的 EventSystem 和 Camera。不难看出，当前场景的构建过程并无太多新奇之处，读者可再次熟悉其中的操作过程。这里也鼓励读者实施进一步探索，在对全部内容有所了解后，可对核心内容进行适当调整。

当选取了 Fuzzy Example 游戏对象后，查看器如图 7.5 所示。

当前实现并非新鲜内容，仅是以一种清晰的方式展示之前的知识点。对于不同的集合。可对其使用 Unity 中的 AnimationCurve，这可视为同一直线可视化操作的一种快捷方式。

然而，并不存在一种直接方式可绘制图中的全部直线，因而需要针对各个集合使用独立的 AnimationCurve，并分别标记为“Critical”“Hurt”“Healthy”。此类曲线包含了内置方法，并在既定点（t）处加以估算。此处，t 表示为时间值，而非 Bob 拥

有的生命值。

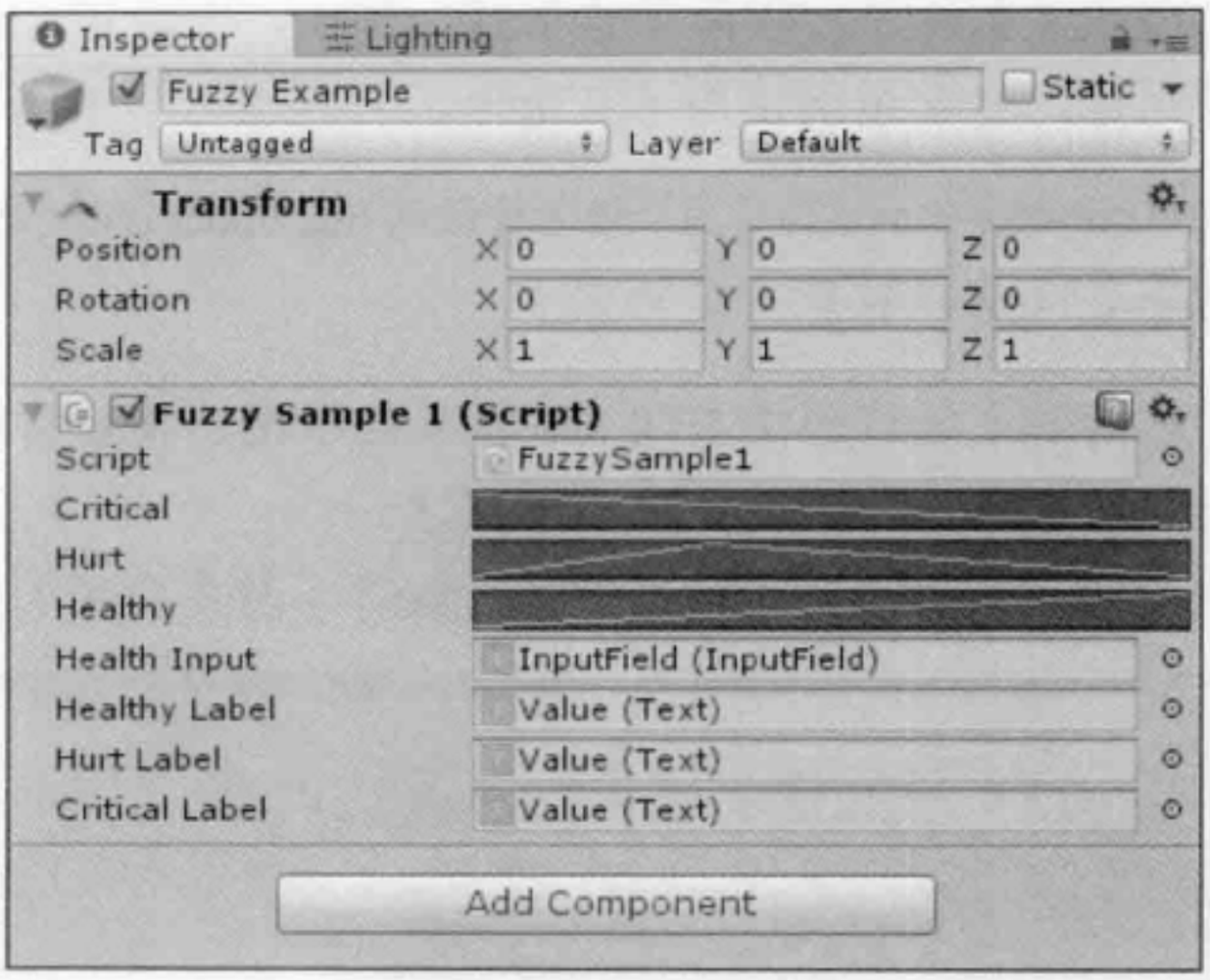

图 7.5

在图 7.5 中，Unity 示例考察 0~100 范围内的 HP，针对此类数据值的编辑处理，对应曲线还提供了简单的用户界面，用户可在查看器中简单地点击曲线，并弹出曲线编辑器窗口。另外，用户还可添加点、移动点或改变切线等，如图 7.6 所示。

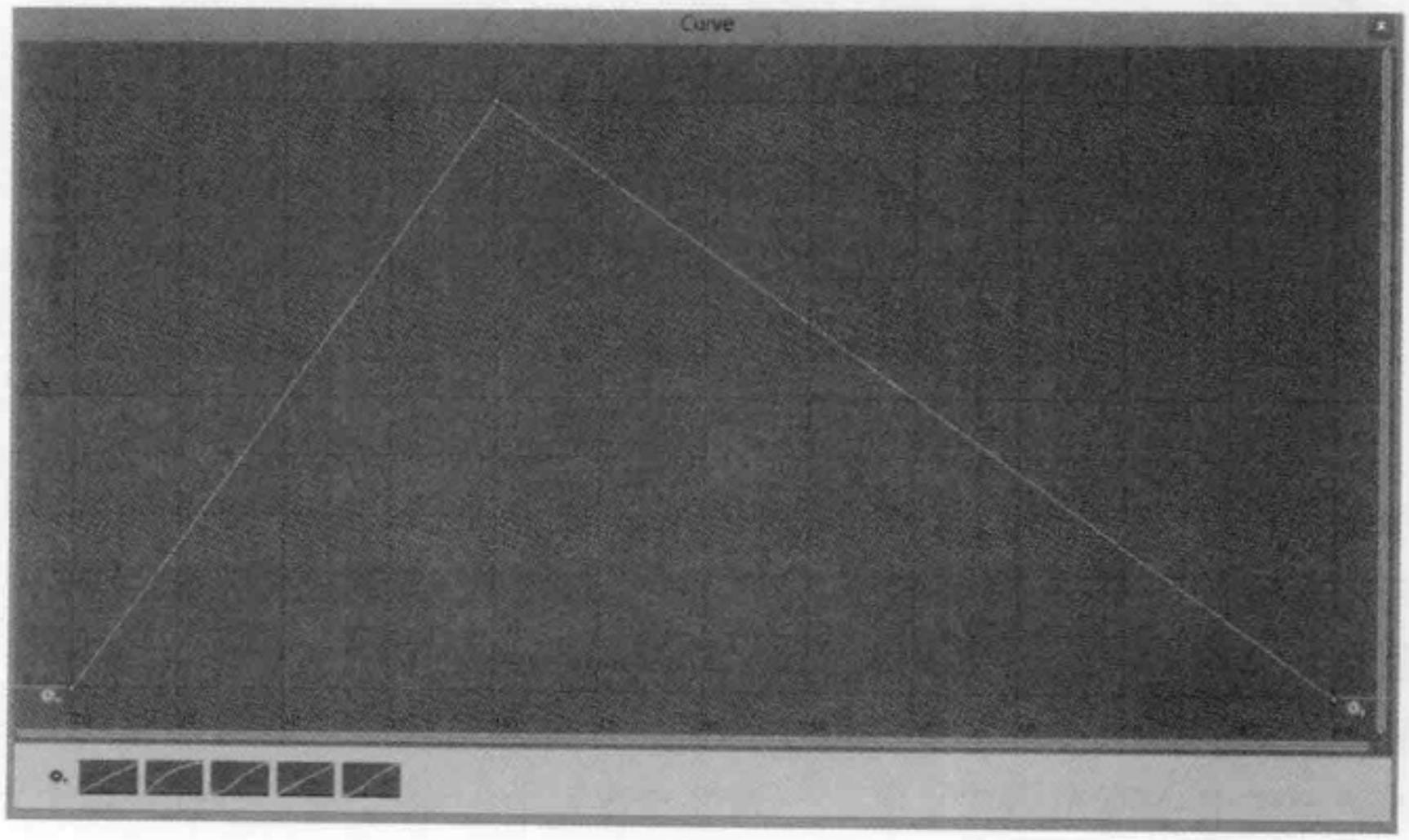

图 7.6

当前示例主要讨论三角形集合，也就是说，针对各集合的线性图。尽管三角形是最为常见的形状，但关注范围不应仅限于该形状。例如，读者还可尝试采用贝尔曲线或梯形。出于简单考虑，当前示例使用了三角形。

技巧：关于 Unity 的 AnimationCurve 编辑器，读者可访问 http://docs.unity3d.com/ScriptReference/AnimationCurve.html 以获取更多内容。

其他字段中的内容表示为代码中不同 UI 元素的引用，稍后将对其加以考查。此类变量名称均具有自解释性质，因而此处不予赘述。

下面讨论场景的构建方式。当对场景进行播放时，游戏视图如图 7.7 所示。

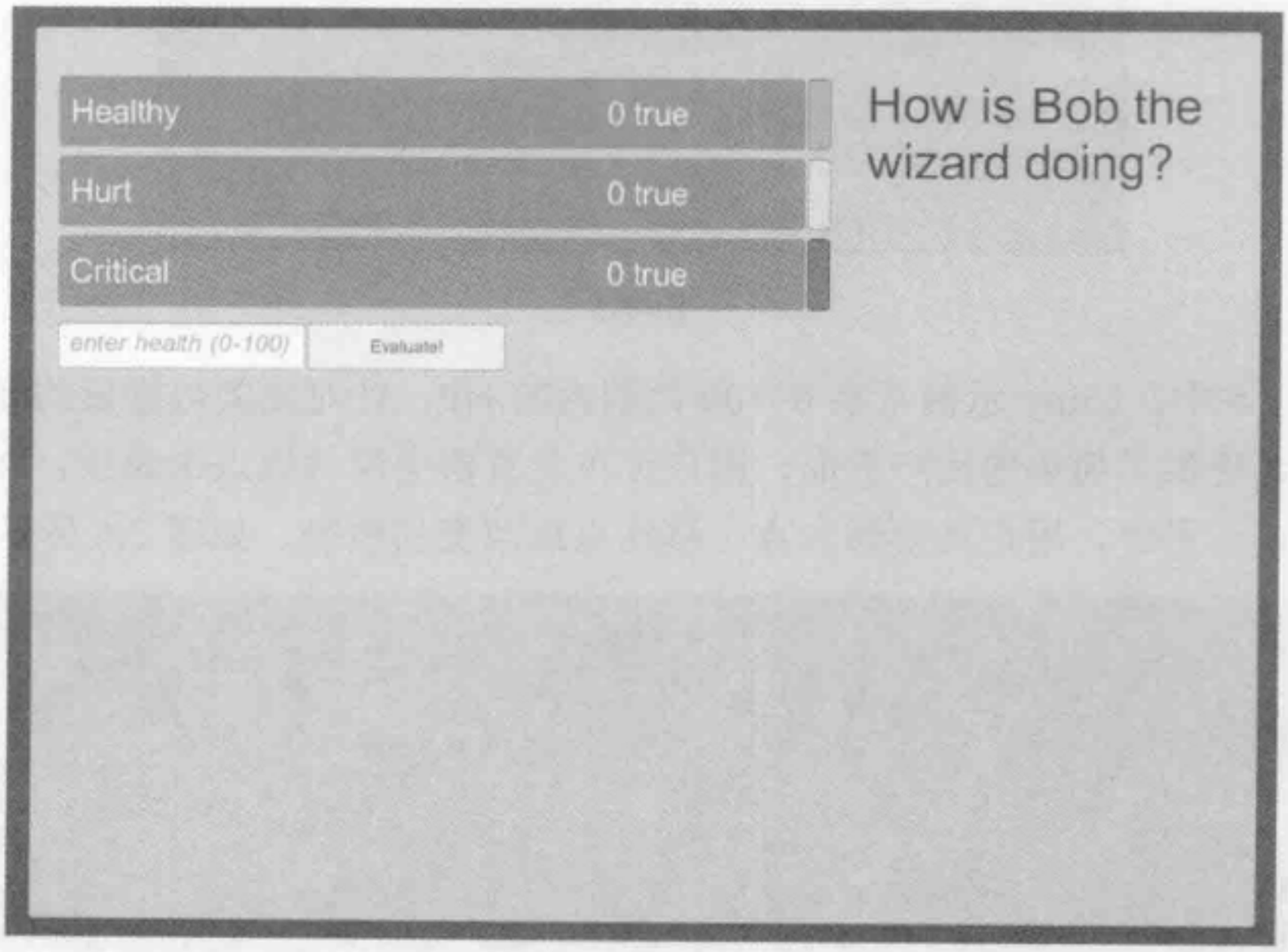

图 7.7

不难发现，图 7.7 中包含了 3 个不同的分组，分别表示 Bob 的各个逻辑结果，即健康状态、受伤状态，以及临界状态。针对各个集合，根据实际估算方式，始于“0 true”的数值经动态调整后可表示为实际的隶属程度。

在输入框中，用户可输入 Bob 的生命值，以供测试使用。需要注意的是，此处应确保输入值位于 0~100 范围内。出于一致性考虑，当前输入 65，随后可按下 Evaluate! 按钮。

在查看曲线后可知，这将生成与前述示意图相同的结果。与测试假设条件相比，游戏程序设计过程中应注意某些重要的细节问题，但可以确定的是，前述逻辑结果已经过测试并得到确认。

待测试过程运行完毕后，游戏场景如图 7.8 所示。

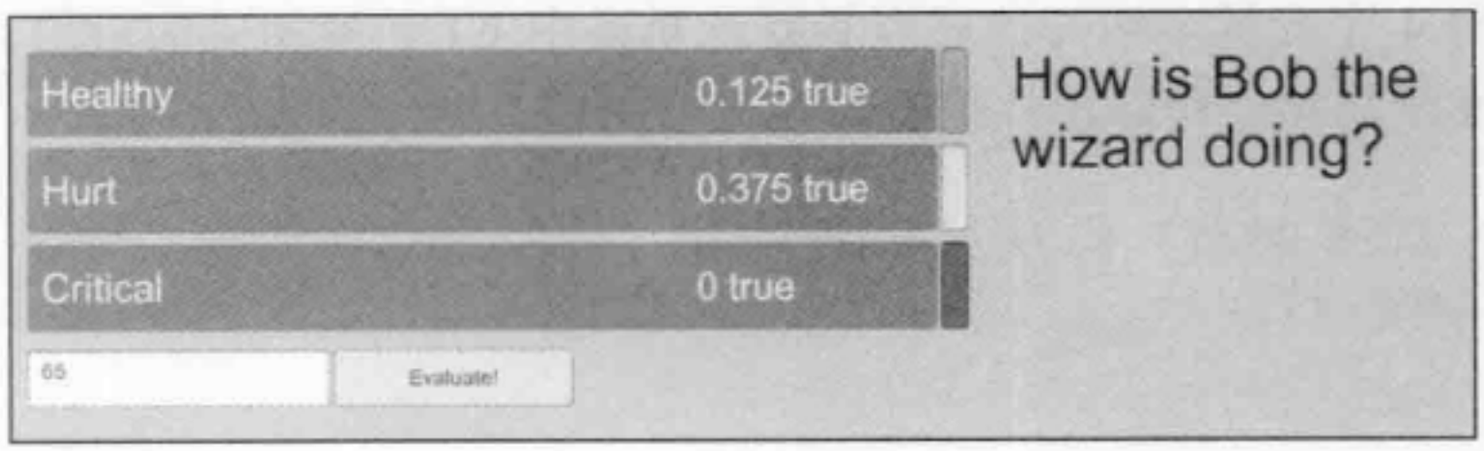

图 7.8

当前生命值定义为 0.125（或 12.5%），损伤值定义为 0.375（或 37.5%）。这里暂且不考虑此类数据，并首先查看对应的代码，如下所示：

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class FuzzySample1 : MonoBehaviour {
    private const string labelText = "{0} true";
    public AnimationCurve critical;
    public AnimationCurve hurt;
    public AnimationCurve healthy;
    public InputField healthInput;

    public Text healthyLabel;
    public Text hurtLabel;
    public Text criticalLabel;

    private float criticalValue = 0f;
    private float hurtValue = 0f;
    private float healthyValue = 0f;
```

代码首先声明了某些变量，其中，labelText 简单地表示为载入标签的常量，并可

用实际值替换{0}。

随后声明了之前讨论的 3 个 AnimationCurve 变量，此类变量表示为 public 类型，以使查看器可对其进行访问，进而实现可视化的编辑操作（尽管也可通过代码生成曲线），这可视为应用过程中的重点内容。最后 3 个变量则表示曲线。

接下来的 4 个变量表示与前述查看器示意图中 UI 元素相关的引用，最后 3 个变量定义为曲线所估算的实际浮点值。

```
    private void Start () {
        SetLabels();
    }
    /*
     * Evaluates all the curves and returns float values
     */
    public void EvaluateStatements() {
        if (string.IsNullOrEmpty(healthInput.text)) {
            return;
        }
        float inputValue = float.Parse(healthInput.text);

        healthyValue = healthy.Evaluate(inputValue);
        hurtValue = hurt.Evaluate(inputValue);
        criticalValue = critical.Evaluate(inputValue);

        SetLabels();
    }
```

Start()方法仅负责更新标签，并执行初始化操作（而非默认文本）。EvaluateStatements()方法较为有趣。首先针对输入字符串检测是否为 null——此处并不打算解析空字符串，否则，函数将直接返回。如前所述，此处无法保证非数值数据的输入行为，因而应避免输入非数值数据，否则用户将会得到错误消息。

对于各个 AnimationCurve 变量，将调用 Evaluate(float t)方法。其中，将采用输入框中的解析值替换 t。在当前运行的示例中，对应值为 65。随后，将再次更新标签，进而显示获取后的数据值。对应代码如下所示：

```
    /*
     * Updates the GUI with the evaluated values based
```

```
     * on the health percentage entered by the
     * user.
     */
    private void SetLabels() {
        healthyLabel.text = string.Format(labelText, healthyValue);
        hurtLabel.text = string.Format(labelText, hurtValue);
        criticalLabel.text = string.Format(labelText, criticalValue);
    }
}
```

代码简单地获取各标签，并通过 labelText 常量的格式化版本替换文本（利用实际值替换{0}）。

7.2.2 扩展集合

需要注意的是，在当前示例中，构建集合的相关值对于 Bob 及其疼痛阈值是唯一的。假设添加另一个精灵巫师角色 Jim，且该角色相对鲁莽。因此，与 Bob 相比，其临界值低于 20 而非 40。这正是模糊逻辑的优势所在，游戏中的各个主体角色可包含定义其集合的不同规则，但系统对此并不关注。相应地，用户可预定义此类规则，或者设置某种等级或随机性，进而确定限定条件，各个单一主体对象将包含独特的行为，并以自身特有的方式响应事件。

另外，并不存在特定规则将集合数量限定为 3 个。读者也可以尝试 3 个或 4 个集合。在模糊逻辑控制器中，全部重点在于确定所到达的正确目标，及其实现方式，且无须关注不同集合的数量，以及系统中现有的多种可能性。

7.2.3 数据的逆模糊化

上述章节讨论了某些观测数据，在模糊逻辑环境下，表示为相对清晰的硬性定义的数据。随后，可将隶属函数赋予集合中，并实现模糊化操作。该处理过程的最后一个步骤是数据的逆模糊化，进而制定某一项决策。对此，可使用下列简单的布尔操作：

```
IF 生命值 IS 临界状态 THEN 施加治愈魔法
```

此时，读者可能会产生如下疑问：这与二元控制器区别何在？为何要使问题复杂化？回忆一下，上述示例曾讨论了模糊信息，如果缺乏模糊控制器的支持，主体角色

如何理解“临界条件”、“受伤状态”以及“生命值”这一类概念？此类抽象概念对于计算机而言意义甚微。

通过模糊逻辑，可使用相应的模糊词语，从中得出相应推论，并执行确实的相关任务，当前示例则是释放治愈魔法。进一步讲，各个主体对象可在个体层次上确定模糊词语的具体含义，实现不可预知的事物，甚至可在多个类似实体对象间完成。

对应处理过程如图 7.9 所示。

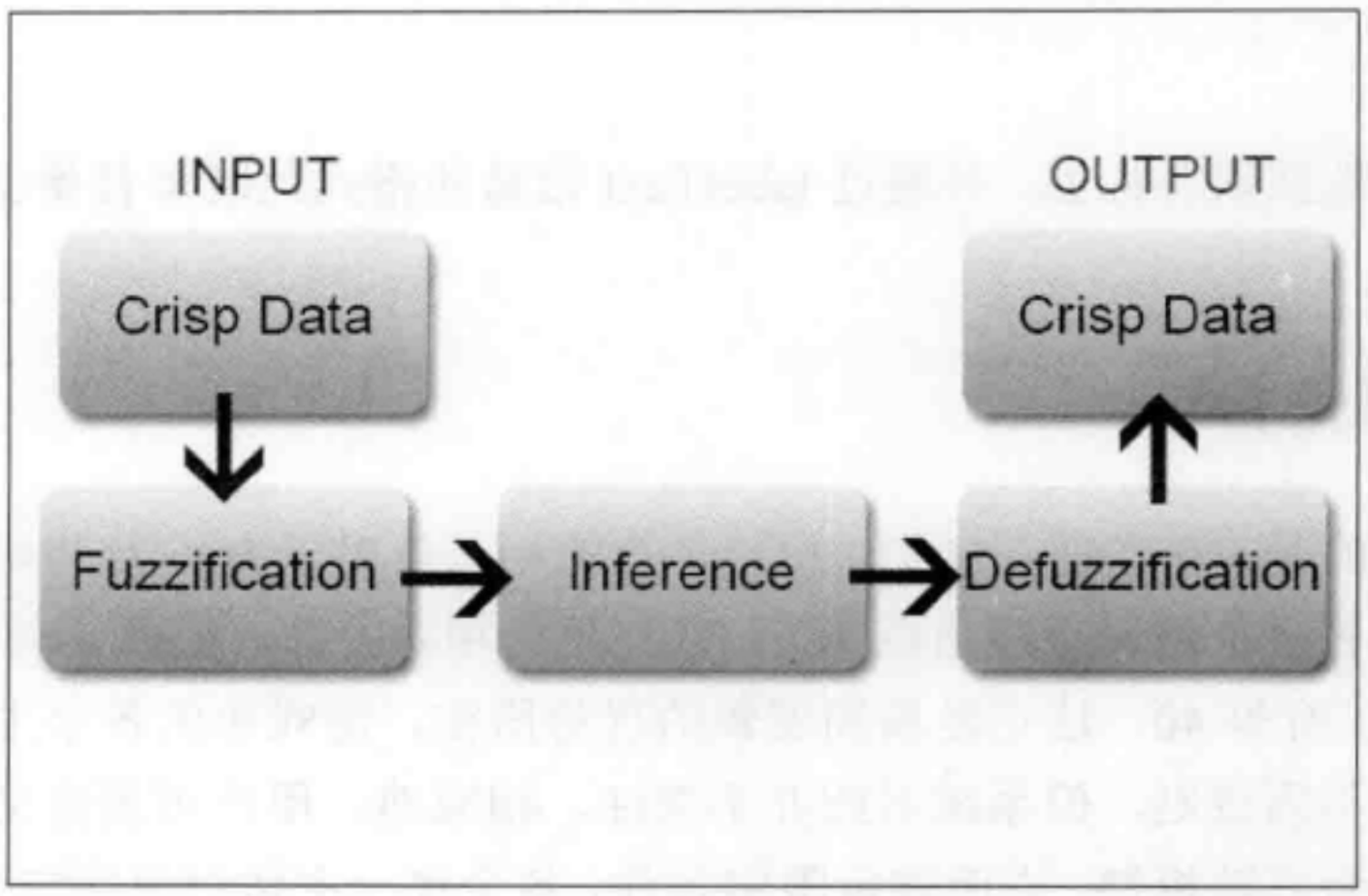

图 7.9

下面讨论计算机设备所能理解的基本事物，即 0 秒和 1 秒，如下所示：

- 首先是观测数据，即特定的具体值。
- 模糊化步骤确定抽象或模糊数据，主体对象据此制定决策。
- 在推理步骤中，主体对象确定数据的含义。根据所提供的规则集确定“正确”的结果，即模拟人类决策过程中的细微差别。
- 逆模糊化步骤采用人类可识别的数据，并将其转换为简单的计算机数据。
- 最后则是观测数据，以供巫师主体对象使用。

7.3　使用观测数据

随后，源自模糊控制器的数据可插入至行为树或有限状态机中。当然，还可整合

多个控制器输出结果，进而制定相关决策。实际上，可利用上述内容实现逼真而有趣的效果（使得魔法师的行为令人更加信服）。图 7.10 显示了模糊逻辑控制器集合，并可用于确定是否施加治愈魔法术。

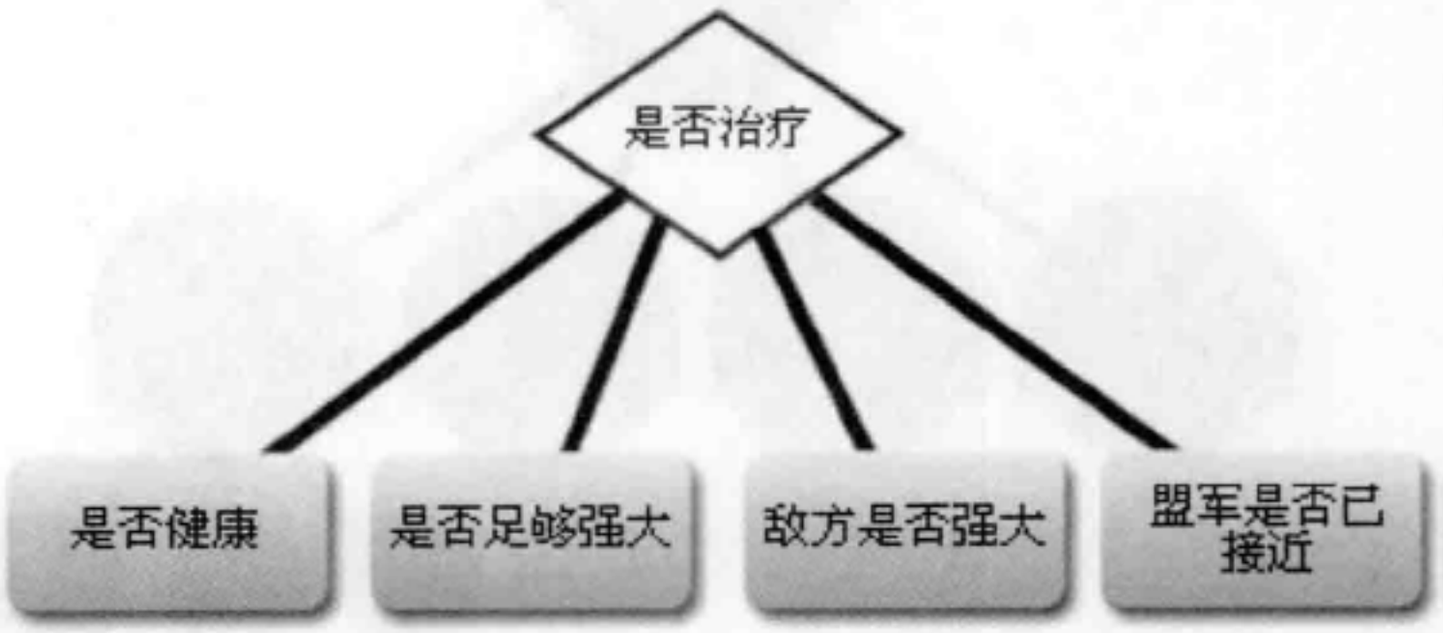

图 7.10

上述内容讨论了角色的生命值问题，针对其他问题，可设定另一个问题集合，且对主体对象自身并不具备太多意义。

例如，玩家是否具有足够的魔法力量？对此，玩家可分别拥有少量、适中以及足够的魔法值。当玩家在游戏中选择相应的法术时，通常会询问此类问题。这里，“足够”表示为一类二元量值，一种可能的情况是，法师包含足够的生命治愈值，但其他法术则包含某些剩余值。下面首先考查一类较为直观的观测数据，即主体对象包含的魔法值，随后可移至模糊逻辑控制器，并在另一端获取某些观测数据。

关于敌方角色的力量值，可能处于薄弱、平均、强大以及不可战胜等状态。同时，用户还可针对模糊逻辑控制器并利用输入数据实现其他创意。例如，可从敌方角色处获取原力量值，并获得玩家防守值和敌方角色攻击力量间的差值，并将结果移至模糊逻辑控制器中。回忆一下，在进入控制器之前，数据的处理方式并无太多限制。

其他问题还包括，盟军是否已向我方靠拢？在第 2 章曾讨论过，对于简单的设计方案，距离检测往往十分有效，但某些时候也需要考虑其他因素，例如路途中的障碍物——盟军是否被紧锁的大门所阻挡，以使其无法接近主体对象？此类问题甚至可作

为逻辑结果嵌套集进行估算。

图 7.11 显示了与最后一个问题相关的嵌套控制器。

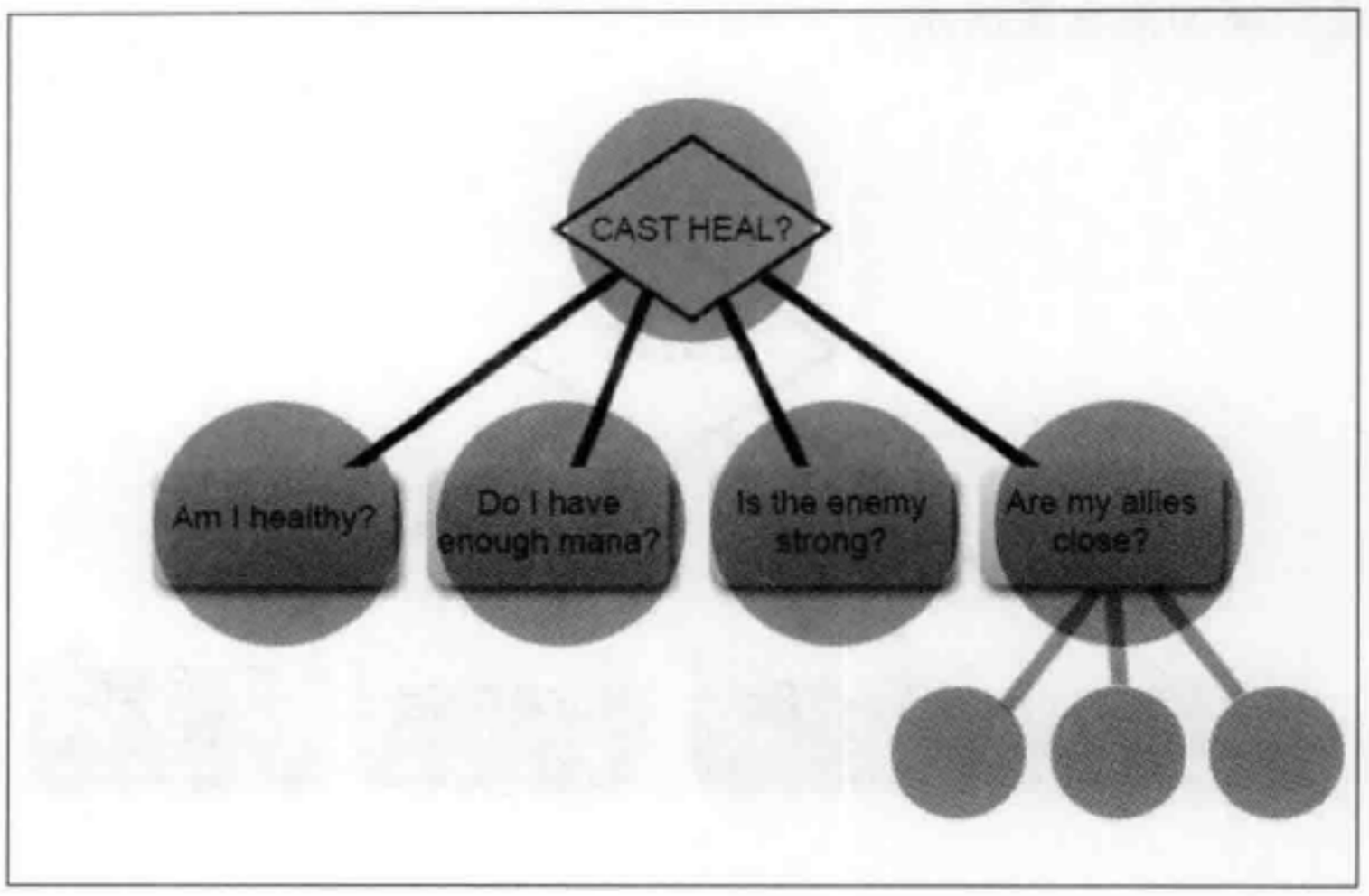

图 7.11

其中包含了与之相应的树状结构，并可通过模糊逻辑构建行为树，进而对各个节点进行估算。通过整合上述两个概念，最终可形成灵活、强大且较为精致的 AI 系统。

如果选择了基于观测输出结果的、相对简单的估算方案，换言之，此处并未采用树形结构或 FSM，而是通过多个布尔操作符确定主体对象的操作内容，对应的伪代码如下所示：

```
IF 生命值 IS 临界状态 AND 魔法值 IS 充沛 THEN 施加治愈魔法
```

同时还可针对非 true 条件进行检测，如下所示：

```
IF 生命值 IS 临界状态 AND 盟军 ARE  NOT 接近 THEN 施加治愈魔法
```

除此之外，还可进一步串接多个条件，如下所示：

```
IF 生命值 IS 临界状态 AND 魔法力量 IS NOT 耗尽 AND 敌方角色 IS 十分强大
THEN 施加治愈魔法
```

通过考查上述简化的逻辑内容，还可发现模糊逻辑的另一个优势——输出观测数据对大量的决策条件予以简化，并将其整合至简化的数据中。

此处无须解析 if/else 语句中的全部可能性，且无须使用大量的 if/else 或 switch 语句，读者可将逻辑部分简单地置于少量且更具实际意义的数据块中。

也就是说，用户无须通过过程方式嵌套全部语句，进而提升代码的阅读难度并降低复用性。作为一类设计模式，基于模糊逻辑控制器的数据抽象机制具有面向对象特征和友好性。

7.4　模糊逻辑的其他应用

模糊数据相对特殊，并可通过本书介绍的多个概念实现“串接”应用方式。下面将考查一系列的模糊逻辑控制器与行为树结构之间的适配方式。相应地，其与 FSM 之间的应用方式也并不十分复杂。

7.4.1　加入其他概念

感知系统也将使用到模糊逻辑。在弱光或低反差环境中，虽然某些观察行为可视为二元条件，但此类条件也可随时转换为模糊条件。例如，读者可能体验过夜间行车，并在远方暗处发现某一奇怪的形状。此时，读者可能会认为是一只猫、垃圾袋或者其他动物，甚至是某种幻觉。相同情形也会出现于声音或气味等现象中。

路径搜索问题需要计算网格区域的行进估值，其中，模糊逻辑控制器可方便地实现模糊化操作，进而可添加更为有趣的运动行为。

又如，Bob 是否应与防守士兵进行较量，进而跨越桥梁并获得路径；或是穿越水面湍急的河流？如果 Bob 擅长游泳且拙于搏斗，则最终答案不言而喻。

7.4.2　创建独特的体验

主体对象还可通过模糊逻辑模拟其内在的个性特征。例如，某些主体对象可能会表现得更为勇敢。除此之外，奔跑速度、行进距离以及个体尺寸均可用于不同主体间的决策制定。

个性特征可应用于敌方角色、盟军、友军、NPC，甚至是游戏规则中。游戏可从玩家进度、体验风格或者关卡进度动态地调整难度，进而提供独特且极具个性化的挑战内容。

模糊逻辑甚至可用于制定特定的游戏规则，例如多玩家游戏场景中的玩家人数、显示于玩家的数据类型，甚至是玩家间的组队方式。适配系统获取玩家统计数据可有效地保持玩家的兴趣度，包括合作环境中的操作风格，以及博弈环境下玩家间的技能水平。

7.5 本 章 小 结

当读者理解了模糊逻辑的基本概念后，其背后所隐藏的内容将变得越发清晰。作为本书中的纯数学概念之一，如果读者对于相关术语了解较少，理解过程也会变得相对艰难。然而，当读者在后续学习过程中面对这一熟悉的环境时，其陌生感将会有所降低，并可在游戏设计中使用这一强大的工具。

本章讨论了模糊逻辑在真实世界中的运用方式，并展示了二元系统所欠缺的模糊化概念。除此之外，本章还通过成员函数、隶属程度以及模糊集等内容实现了模糊逻辑控制器。最后，本章还介绍了结果数据的各种应用方式，从而使得主体对象具有更加独特的行为方式。

第 8 章将讨论多种概念间的整合方式。

第8章 整合过程

上述章节讨论了实现 Unity 游戏 AI 的必要工具，这一内容贯穿了全书。需要注意的是，全部概念和模式均独立成章，经适当整合后可实现期望的 AI 行为。本章主要讨论简单的坦克攻防游戏，实现之前介绍的某些相关概念，以及一款整合型游戏。需要说明的是，该游戏更像是一幅设计蓝图，读者可对其进行适当扩展并尝试各种体验功能。本章主要涵盖下列内容：

- 对前述简单项目中的系统进行整合。
- 创建 AI 高塔主体对象。
- 创建 NavMeshAgent 坦克对象。
- 构建场景环境。
- 对示例场景进行测试。

8.1 制定规则

当前游戏内容相对简单，而真实的游戏逻辑，例如生命值、破坏值以及通关条件，则留予读者完成，此处主要讨论各个组件的构建过程，进而实现一款坦克攻防游戏。

当定义主体对象所需的逻辑和行为类型时，需要对游戏规则实现具体化操作，而非简单的理念内容。当然，在实现不同特性过程中，此类规则还可不断进行调整，但强调前述内容所讨论的概念集将有助于选择最佳实现工具。

当前游戏与传统的塔防游戏稍有不同，读者不必通过高塔以阻止来犯的敌方角色，而是令坦克对象穿越高塔的火力网。在坦克穿越迷宫型关卡时，途中的高塔会向坦克发射飞弹。为了使坦克成功地穿越火线，读者可使用下列两个功能项。

- 加速功能：该功能可在短期内令坦克对象以倍速方式运动。这对于躲避飞弹袭击十分有用。
- 防御功能：该功能可在短期内在坦克周围形成防护罩，以阻挡来袭的飞弹。

当前示例采用了有限状态机实现高塔对象，其原因在于，这一类对象包含有限数

量的状态，且无须使用到相对复杂的行为树。另外，高塔对象需要对周边环境进行侦察，特别地，需要判断附近的坦克对象是否位于射程范围内，并对其展开攻击。对此，可采用球体触发器对高塔对象的视域以及感知系统建模。相应地，坦克对象应可在场景中实现自身的导航行为，因而此处使用 NavMesh 和 NavMeshAgent 对其加以定义。

8.2　创建高塔对象

在本章的示例项目中，Prefabs 文件夹内包含了 Tower 预制组件。高塔对象自身的实现方式较为简单，并由一组图元构成，如图 8.1 所示。

其中，枪管附着于球体上，当追踪玩家时可沿其轴向自身旋转，并在目标方向上射击。当坦克对象远离后，高塔对象则无法继续跟踪，或者无法实现自身的重定位。

在当前示例场景中，关卡中设置了多个高塔对象，作为预制组件，这一类对象可方便地复制、移动，并在关卡间予以复用。其构建过程并不特别复杂，对应的层次结构如图 8.2 所示。

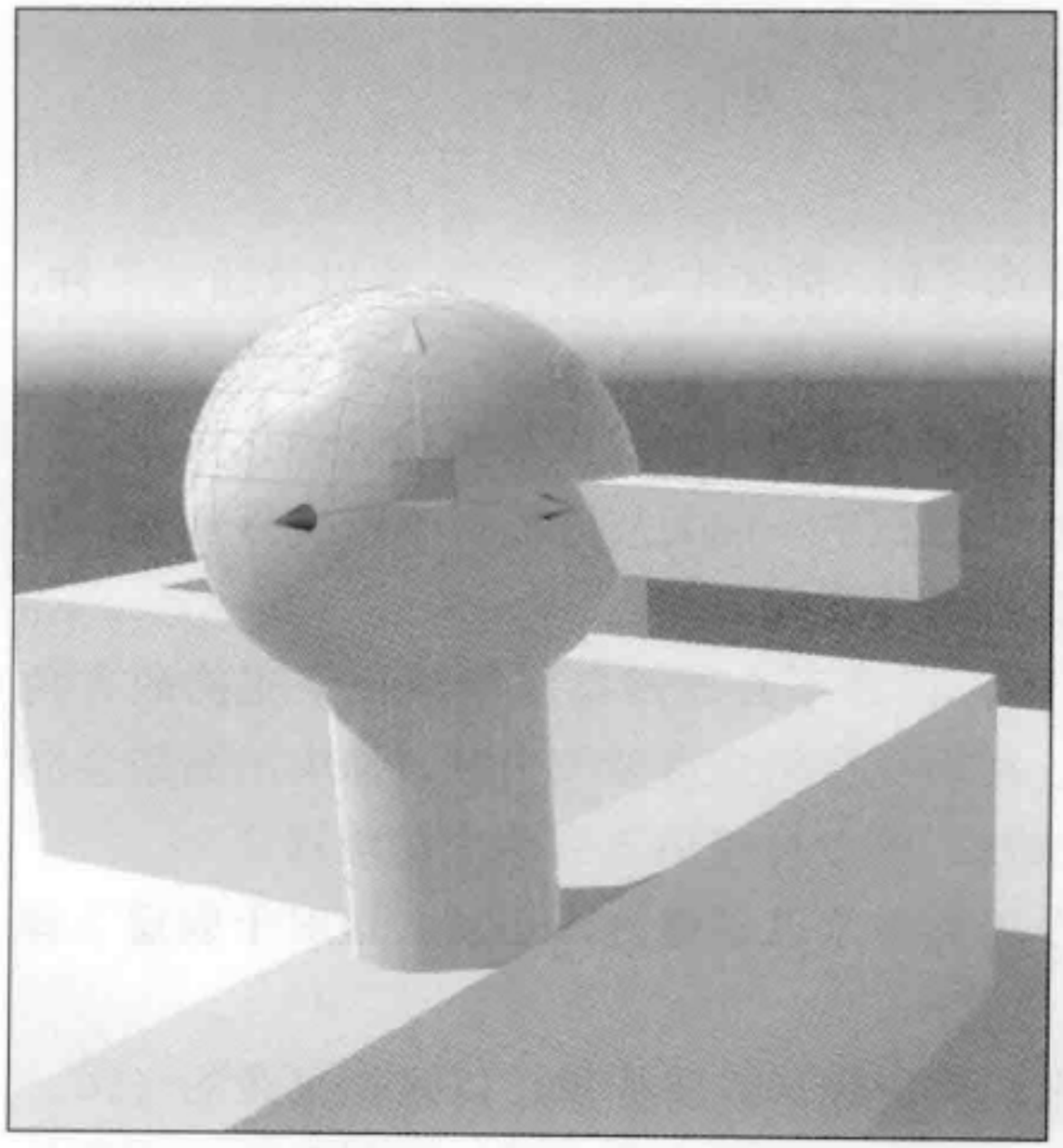

图 8.1

图 8.2

其中各项内容如下所示。

- Tower：从技术角度上讲，这可视为高塔对象的基础设施——包含了相关组件的圆柱体且不具备实际功能。
- Gun：高塔（包含枪管）上的球体对象，作为高塔对象上的部分设施，可移动并跟踪玩家。
- Barrel 和 Muzzle：枪口位于枪管的末端，并作为子弹的发射点。

作为高塔的主要工作组件，下面对枪管稍加讨论。在选取了枪管对象后，查看器如图 8.3 所示。

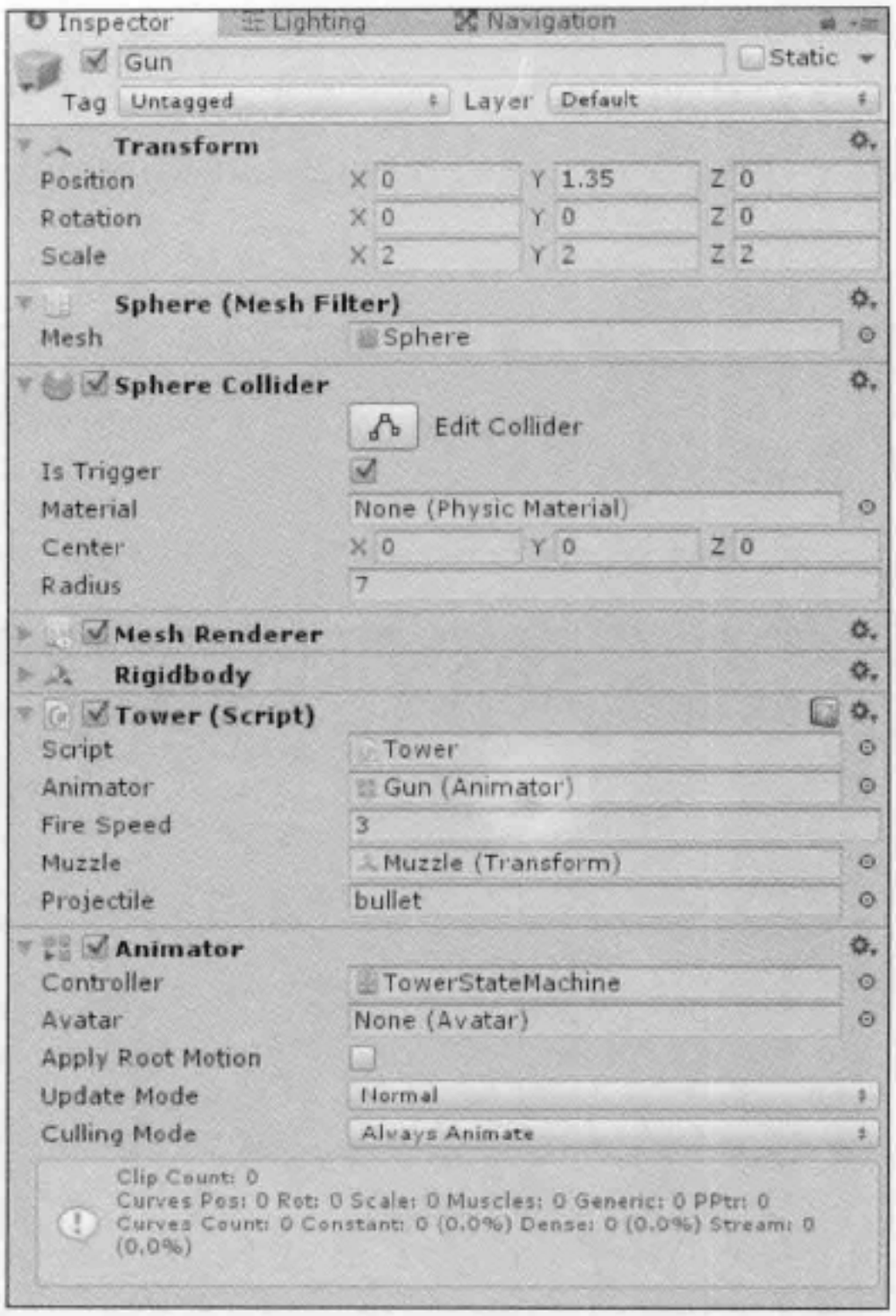

图 8.3

此处，查看器中包含了多项内容，下面对影响逻辑内容的各个组件注意加以考查，如下所示。

- Sphere Collider：实际上，这可视为高塔的防御范围。一旦坦克对象进入该球

体，高塔即可侦察到这一行为，锁定坦克对象后并对其予以射击，因而定义了高塔对象的感知系统的实现方式。需要注意的是，当前半径设置为 7。虽然该值可根据个人喜好进行调整，但当前值相对公平。另外，需要注意的是，此处选中了 Is Trigger 复选框，也就是说，该球体并非导致碰撞行为，而是引发某种触发器事件。

- Rigidbody：该组件供碰撞器使用，无论对象是否处于运动状态，其原因在于，Unity 并不会向静态游戏对象传递碰撞或触发器事件，除非其包含刚体组件。
- Tower：即高塔对象的逻辑脚本，并与状态机和状态机行为协同工作，稍后将对其予以详细分析。
- Animator：即高塔对象的状态机，该组件并不会处理动画问题。

在考查高塔对象的驱动代码之前，首先对状态机加以简要回顾。对应状态机并不复杂，如图 8.4 所示。

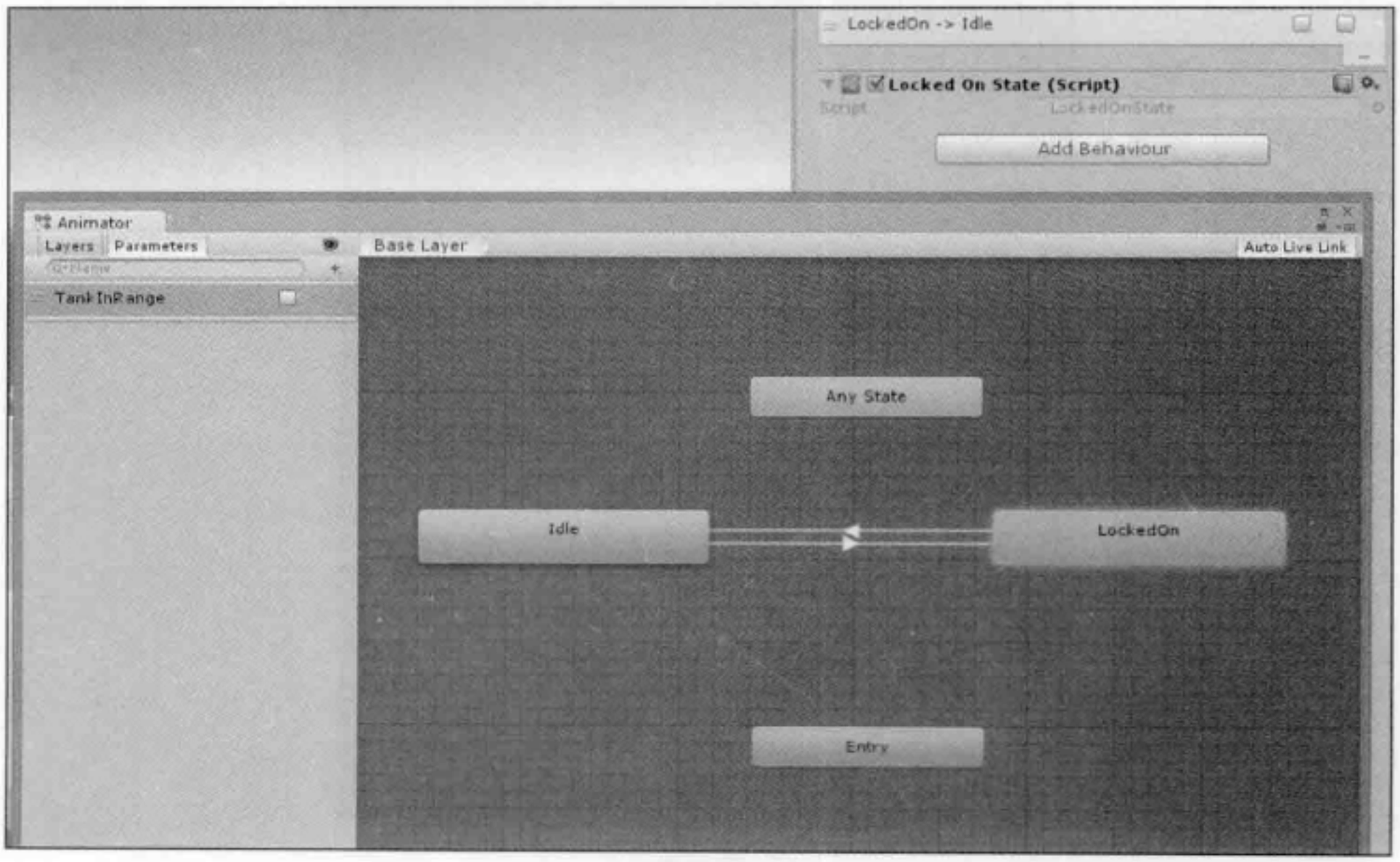

图 8.4

此处关注两种状态，即 Idle（默认状态）和 LockedOn 状态。当 TankInRange 布尔变量设置为 true 时，将导致 Idle 与 LockedOn 之间的转换；相应地，当该布尔变量设置为 false 时，将产生逆转换。

LockedOn 包含了与其绑定的 StateMachineBehaviour 类，如下所示：

```
using UnityEngine;
using System.Collections;

public class LockedOnState : StateMachineBehaviour {

    GameObject player;
    Tower tower;

    //OnStateEnter is called when a transition starts and the state
machine starts to evaluate this state
    override public void OnStateEnter(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex) {
        player = GameObject.FindWithTag("Player");
        tower = animator.gameObject.GetComponent<Tower>();
        tower.LockedOn = true;
    }

    //OnStateUpdate is called on each Update frame between
OnStateEnter and OnStateExit callbacks
  override public void OnStateUpdate(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex) {
        animator.gameObject.transform.LookAt(player.transform);
  }

    //OnStateExit is called when a transition ends and the state
machine finishes evaluating this state
    override public void OnStateExit(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex) {
        animator.gameObject.transform.rotation = Quaternion.identity;
        tower.LockedOn = false;
    }
}
```

当进入当前状态且 OnStateEnter 被调用时，将得到指向当前玩家的引用。在所提供的示例中，玩家对象标记为“Player”，因而可通过 GameObject.FindWithTag 获取指向该对象的引用。随后，可获取与高塔预制组件绑定的、Tower 组件的引用，并将

其 LockedOn 布尔变量设置为 true。

一旦处于当前状态，OnStateUpdate 方法将在每帧内被调用。在该方法内，将通过提供的 Animator 引用获取指向的 Gun GameObject 引用（绑定了 Tower 组件）。通过指向枪械的引用，即可采用 Transform.LookAt 方法令其对坦克对象进行跟踪。

提示： 作为一种替代方法，鉴于Tower的布尔变量LockedOn可设置为true，因而对应逻辑可在Tower.cs脚本中进行处理。

最后，当退出当前状态时，OnStateExit 方法将被调用。该方法执行某些清空操作，例如，枪支的旋转位置将被重置，进而表明不再对玩家予以跟踪。除此之外，还可将 Tower 的 LockedOn 布尔变量重置为 false。

不难发现，StateMachineBehaviour 与 Tower.cs 脚本相互作用。该脚本的具体内容如下所示：

```
using UnityEngine;
using System.Collections;

public class Tower : MonoBehaviour {
    [SerializeField]
    private Animator animator;

    [SerializeField]
    private float fireSpeed = 3f;
    private float fireCounter = 0f;
    private bool canFire = true;

    [SerializeField]
    private Transform muzzle;

    [SerializeField]
    private GameObject projectile;

    private bool isLockedOn = false;

    public bool LockedOn {
        get { return isLockedOn; }
```

```
        set { isLockedOn = value; }
    }
```

代码首先声明了所需的变量和属性。

另外，此处还需要使用到一个指向状态机的引用，这也是 Animator 变量出现之处。fireSpeed、fireCounter 和 canFire 变量则与高塔对象的射击逻辑有关，稍后将查看其具体的工作方式。

如前所述，枪口定义了子弹的射击位置，相应的弹药表示为需要实例化的预制组件。

最后，isLockedOn 将通过 LockedOn 进行设置。通常情况下，除非需要显式地将数据值声明为 public 类型，否则此类变量一般声明为 private。因此，此处 isLockedOn 定义为 private 类型，并提供了一个属性，以对其进行远程访问（当前操作源自 LockedOnState 行为）。

```
    private void Update() {
        if (LockedOn && canFire) {
            StartCoroutine(Fire());
        }
    }

    private void OnTriggerEnter(Collider other) {
        if (other.tag == "Player") {
            animator.SetBool("TankInRange", true);
        }
    }

    private void OnTriggerExit(Collider other) {
        if (other.tag == "Player") {
            animator.SetBool("TankInRange", false);
        }
    }

    private void FireProjectile() {
        GameObject bullet = Instantiate(projectile, muzzle.position,
muzzle.rotation) as GameObject;
        bullet.GetComponent<Rigidbody>().AddForce(muzzle.forward *
300);
```

```
    }

    private IEnumerator Fire() {
        canFire = false;

        FireProjectile();
        while (fireCounter < fireSpeed) {
            fireCounter += Time.deltaTime;
            yield return null;
        }
        canFire = true;
        fireCounter = 0f;
    }
}
```

至此，高塔对象中的相关逻辑暂告一段落。Update 循环检测了两项内容，即是否锁定相关对象，以及是否可执行射击操作。如果两个条件均满足，则调用 Fire()协同例程。在返回至 OnTrigger 消息之前，需要考察 Fire()方法为何定义为协同例程。

提示： 协同例程这一类概念颇具技巧性。关于协同例程的应用方式，读者可阅读Unity的相关文档，对应网址为http://docs.unity3d.com/Manual/Coroutines.html。

此处并不希望高塔对象无休止地对坦克进行射击，对此，可采用之前定义的某些变量，并在每次射击之间创建一个缓冲时间间隔。待调用 FireProjectile()方法并将 canFire 设置为 false 后，可在 0~fireSpeed 之间设置一个计数器，随后再次将 canFire 设置为 true。FireProjectile()方法负责处理弹药的实例化操作，并通过 Rigidbody.AddForce 沿所指方向射击。稍后将考查子弹的操作逻辑。

最后是两个 OnTrigger 事件，分别对应于对象进入/驶离触发器时的事件。对此，读者可回顾一下针对状态机转换驱动的 TankInRange 布尔变量。当进入触发器时，该变量设置为 true；而离开触发器时，该变量将设置回 false。实际上，当坦克对象进入枪械的视见球体时，该对象将被锁定；相应地，当坦克驶离该区域后，锁定行为也将随之被释放。

回顾查看器中的 Tower 组件时将会发现，名为 bullet 的预制组件将赋予 projectile 变量，该组件位于示例项目的 Prefabs 文件夹中，具体内容如图 8.5 所示。

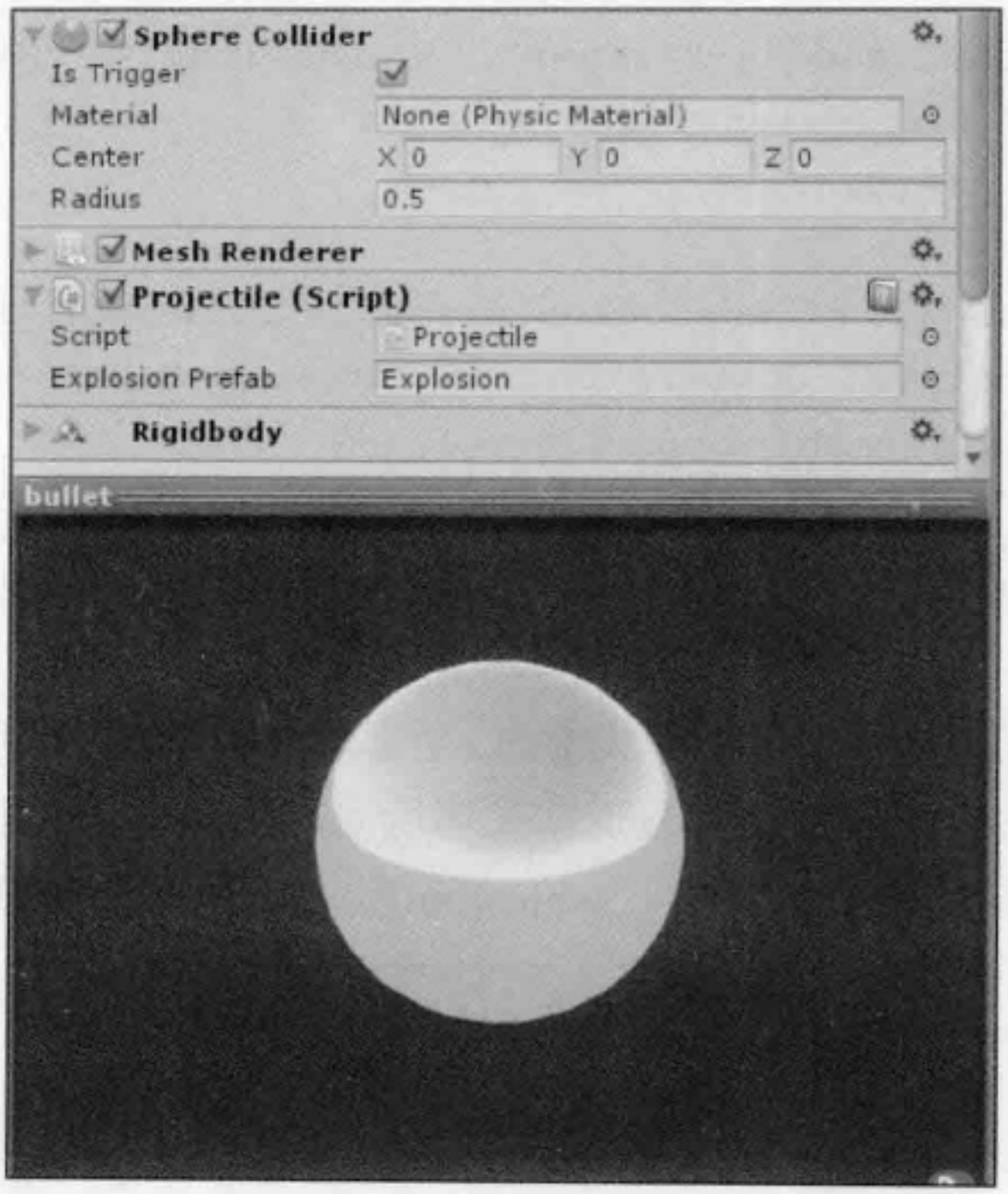

图 8.5

这里，子弹对象仅简单地表示为一个球体，并存在一个与其绑定的球体碰撞器。再次说明，此处需确保 IsTrigger 为 true，且包含一个与其绑定的 Rigidbody（禁用 gravity 项）。除此之外，还需要包含一个与子弹预制组件绑定的 Projectile 组件，并用于处理碰撞逻辑，如下所示：

```
using UnityEngine;
using System.Collections;

public class Projectile : MonoBehaviour {

    [SerializeField]

   private GameObject explosionPrefab;

void Start () { }

   private void OnTriggerEnter(Collider other) {
```

```
        if (other.tag == "Player" || other.tag == "Environment") {
            if (explosionPrefab == null) {
                return;
            }
            GameObject explosion = Instantiate(explosionPrefab,
transform.position, Quaternion.identity) as GameObject;
            Destroy(this.gameObject);
        }
    }
}
```

上述脚本内容较为直观，在当前关卡内，全部地面和墙体均标记为“Environment”，OnTriggerEnter 方法检测子弹是否与玩家或环境间产生碰撞。若是，则实例化 explosion 预制组件，并于随后销毁子弹对象。explosion 预制组件如图 8.6 所示。

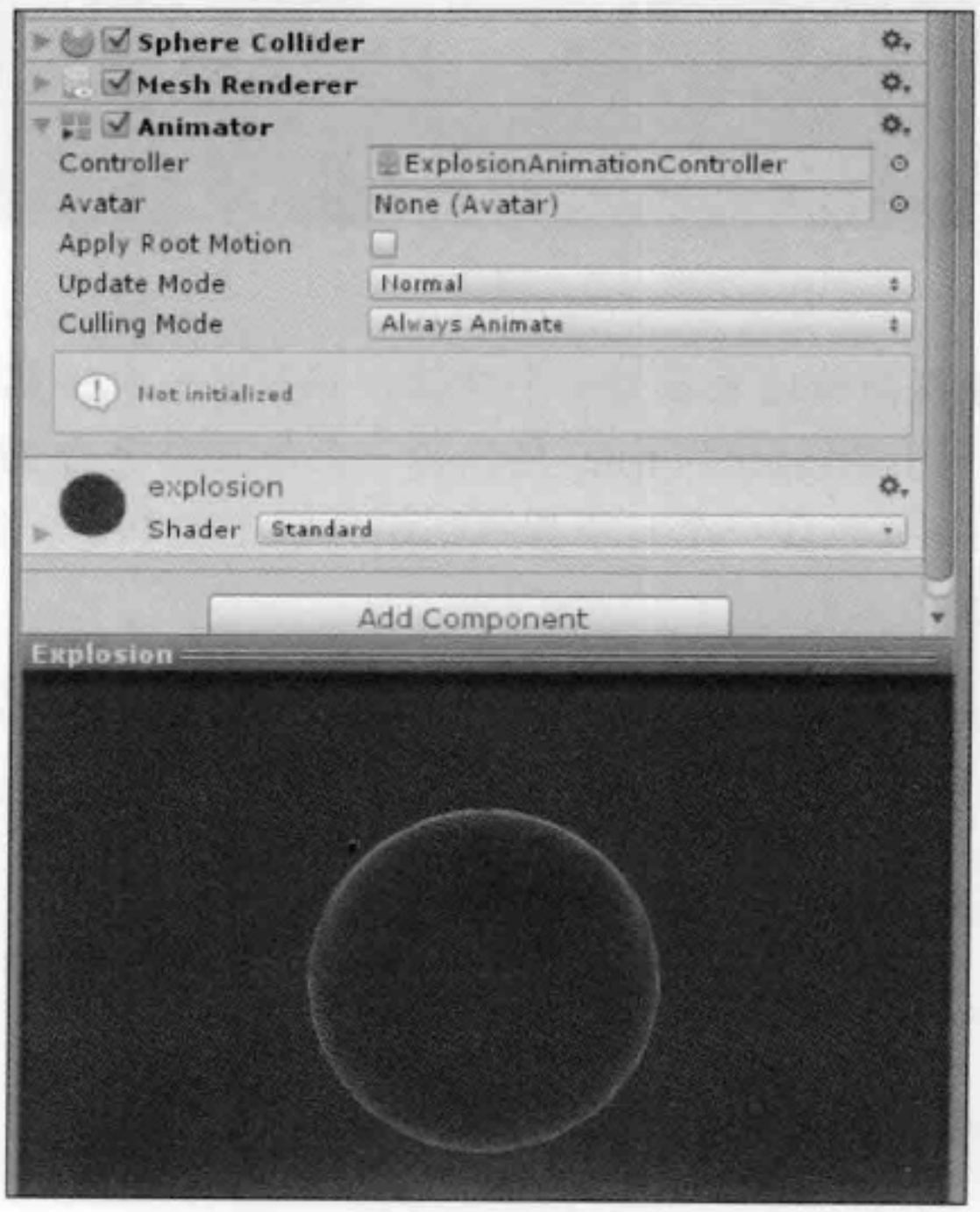

图 8.6

通过观察可知，此处存在一个类似的游戏对象，并包含一个球体碰撞器（IsTrigger

设置为 true）。其中的主要差别在于 animator 组件，当 explosion 对象被实例化后，其行为将呈现为爆炸状态。当爆炸状态结束后，可通过状态机销毁该实例对象。animation 控制器如图 8.7 所示。

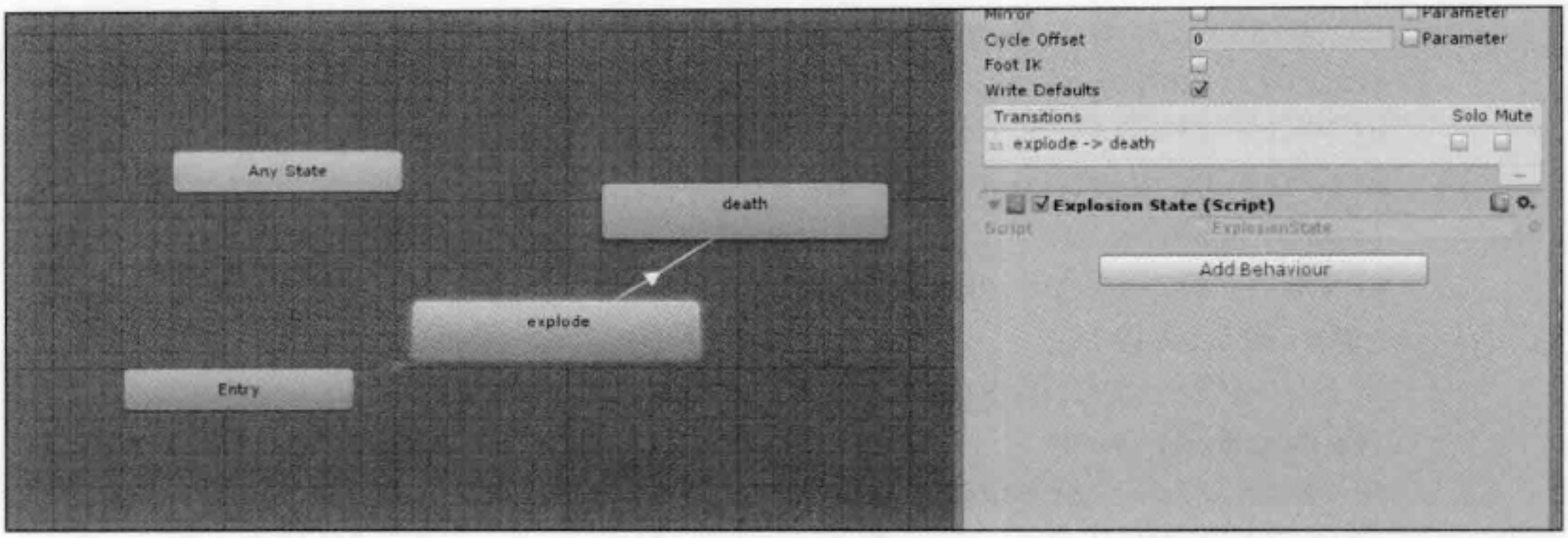

图 8.7

explode 状态包含了绑定于其上的某一行为，其对应代码如下所示：

```
//OnStateExit is called when a transition ends and the state machine
finishes evaluating this state
    override public void OnStateExit(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex) {
        Destroy(animator.gameObject, 0.1f);
    }
```

当退出当前状态时，将销毁实例化对象，这一操作出现于动画结束时。

提示： 读者可通过自己的游戏逻辑进一步丰富游戏内容，并可触发任意附加效果，例如爆炸和环境粒子等内容。

8.3 构建坦克对象

示例项目中还涉及了坦克预制组件，在 Prefabs 文件夹中，该组件称作 Tank。

坦克自身表示为简单的主体对象，其唯一目标是穿越迷宫。如前所述，坦克对象应在玩家帮助下穿越高塔所布置的火线。

针对绑定于预制组件上的 Tank.cs 组件，下列代码显示了其中的内容：

```
using UnityEngine;
using System.Collections;

public class Tank : MonoBehaviour {
    [SerializeField]
    private Transform goal;
    private NavMeshAgent agent;
    [SerializeField]
    private float speedBoostDuration = 3;
    [SerializeField]
    private ParticleSystem boostParticleSystem;
    [SerializeField]
    private float shieldDuration = 3f;
    [SerializeField]
    private GameObject shield;

    private float regularSpeed = 3.5f;
    private float boostedSpeed = 7.0f;
    private bool canBoost = true;
    private bool canShield = true;
```

其中多个值需要进行调整，因而首先声明对应的变量，并于随后对关联状态进行设置，如下所示：

```
    private bool hasShield = false;
private void Start() {
        agent = GetComponent<NavMeshAgent>();
        agent.SetDestination(goal.position);
    }

    private void Update() {
        if (Input.GetKeyDown(KeyCode.B)) {
            if (canBoost) {
                StartCoroutine(Boost());
            }
        }
        if (Input.GetKeyDown(KeyCode.S)) {
```

```
            if (canShield) {
                StartCoroutine(Shield());
            }
        }
    }
```

Start 方法负责执行坦克对象的构建工作，获取 NavMeshAgent 组件，并将其方向设置为目标方向，稍后将对此加以深入讨论。

针对坦克对象的各种能力，Update 方法用于获取输入内容，并将 B 键映射为加速功能，将 S 键映射为防御功能，这与高塔对象的射击行为有些类似，可通过协同例程加以实现，如下所示：

```
    private IEnumerator Shield() {
        canShield = false;
        shield.SetActive(true);
        float shieldCounter = 0f;
        while (shieldCounter < shieldDuration) {
            shieldCounter += Time.deltaTime;
            yield return null;
        }
        canShield = true;
        shield.SetActive(false);
    }

    private IEnumerator Boost() {
        canBoost = false;
        agent.speed = boostedSpeed;
        boostParticleSystem.Play();
        float boostCounter = 0f;
        while (boostCounter < speedBoostDuration) {
            boostCounter += Time.deltaTime;
            yield return null;
        }
        canBoost = true;
        boostParticleSystem.Pause();
        agent.speed = regularSpeed;
    }
```

上述两项行为功能间彼此类似。shield 变量负责开启/禁用游戏对象 shield 功能，并对应于查看器中的某一个变量中。待 shieldDuration 时间值后，可将其关闭，玩家在后续操作中还可再次使用这一功能。

Boost 代码中的主要差别在于，加速功能调用粒子系统（通过查看器赋值）上的 Play 方法，并将 NavMeshAgent 的速度设置为倍速，随后在该功能消失时对其进行重置。

提示：读者可尝试添加新的功能项，并可通过这一相对直观的模式在自己的项目中实现新的功能。除此之外，读者还可添加附加逻辑，进而对防御和加速功能进行自定义。

示例场景包含了坦克对象实例，且全部变量属性均已设置完毕。示例场景中坦克对象的查看器如图 8.8 所示。

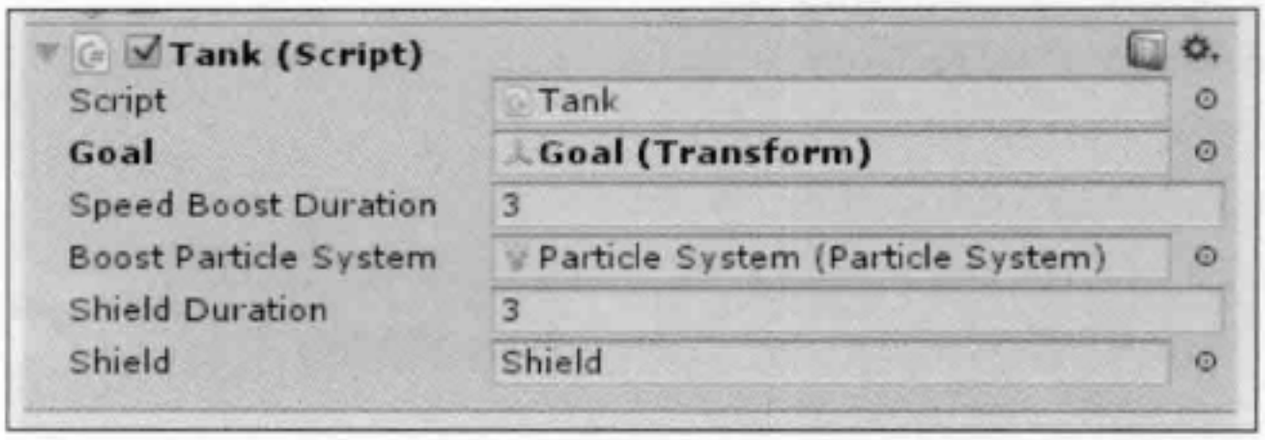

图 8.8

其中，Goal 变量赋予了包含相同名称的转换，且位于迷宫场景的末端。另外，还可适当调整各项功能的时长（默认值为 3）。同时，还可交换各项功能间的设计素材，并设置用于加速功能或防御时游戏对象的粒子系统。

最后一段代码用于驱动相机对象。此处需要相机跟随玩家运动，对应代码如下所示：

```
using UnityEngine;
using System.Collections;

public class HorizontalCam : MonoBehaviour {
    [SerializeField]
    private Transform target;

    private Vector3 targetPositon;

    private void Update() {

        .
```

```
            targetPositon = transform.position;
            targetPositon.z = target.transform.position.z;
            transform.position = Vector3.Lerp(transform.position,
targetPositon, Time.deltaTime);
        }
}
```

不难发现，可简单地将相机位置设置为全部轴向上的当前位置，并于随后将目标位置的 z 轴设置为当前目标。在查看器中，这被设置为坦克对象的转换项。在每帧中，可采用线性插值（Vector3.Lerp）计算，进而在当前位置和目标位置之间实现相机对象的平滑移动。

8.4 构建场景环境

由于坦克对象使用了 NavMeshAgent 组件在环境中行进，为了使烘焙操作（参见第 4 章）正常进行，需要使用静态游戏对象构建场景。在迷宫结构中，高塔对象的设置应相对合理，以使坦克可自由穿行于其中。迷宫的布局方式如图 8.9 所示。

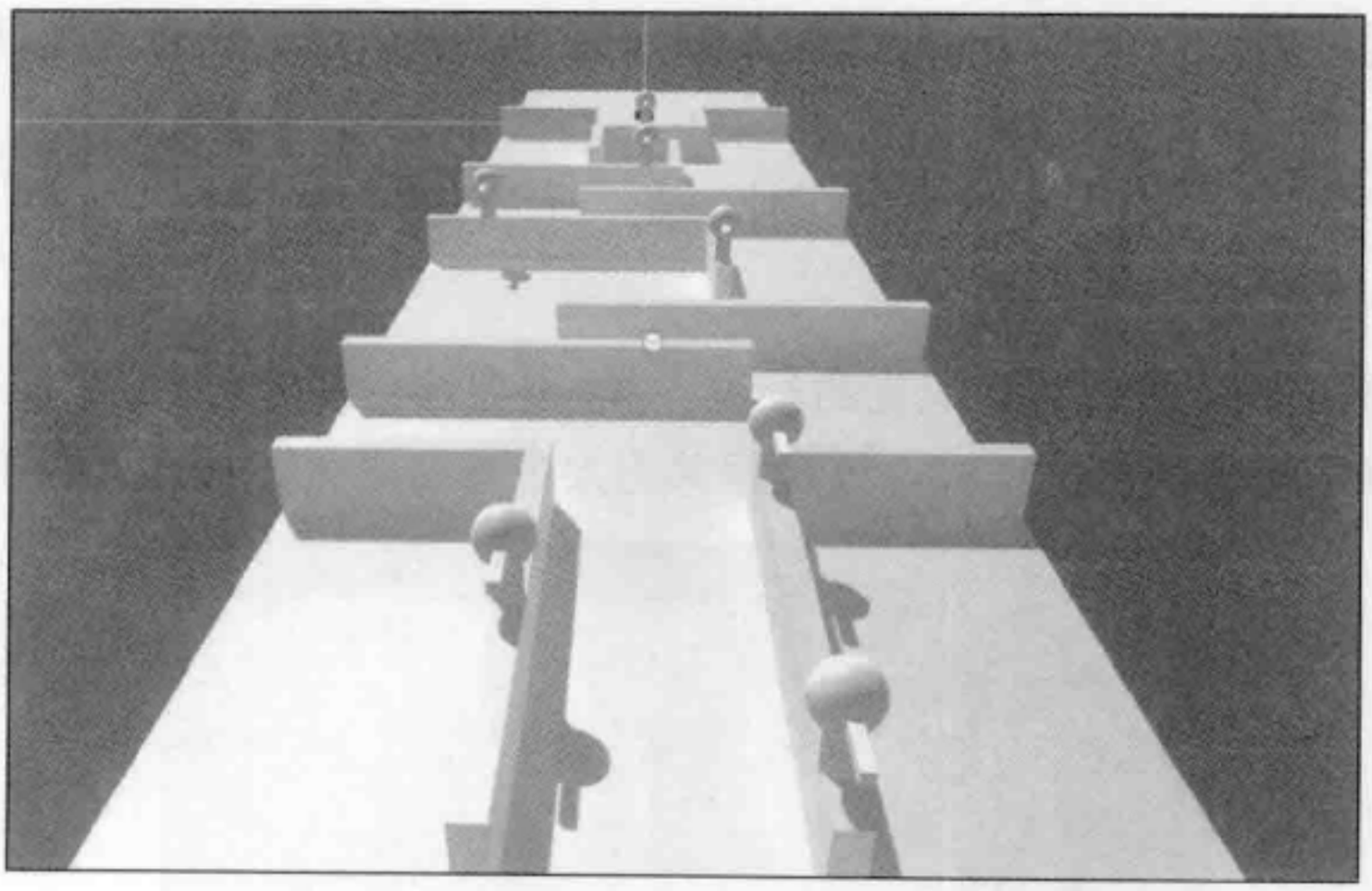

图 8.9

通过观察可知，迷宫中设置了多个高塔，其位置多变以对坦克对象实施有效的侦察。另外，为了避免坦克对象与墙体碰撞，可根据个人喜好调整导航窗口中的设置数

据。默认情况下，示例场景中主体对象的半径范围设置为 1.46，步进高度设置为 1.6。关于这一数值的设置问题，并不存在硬性规定，对应结果仅是通过反复尝试而得到。

待 NavMesh 烘焙完毕后，最终结果如图 8.10 所示。

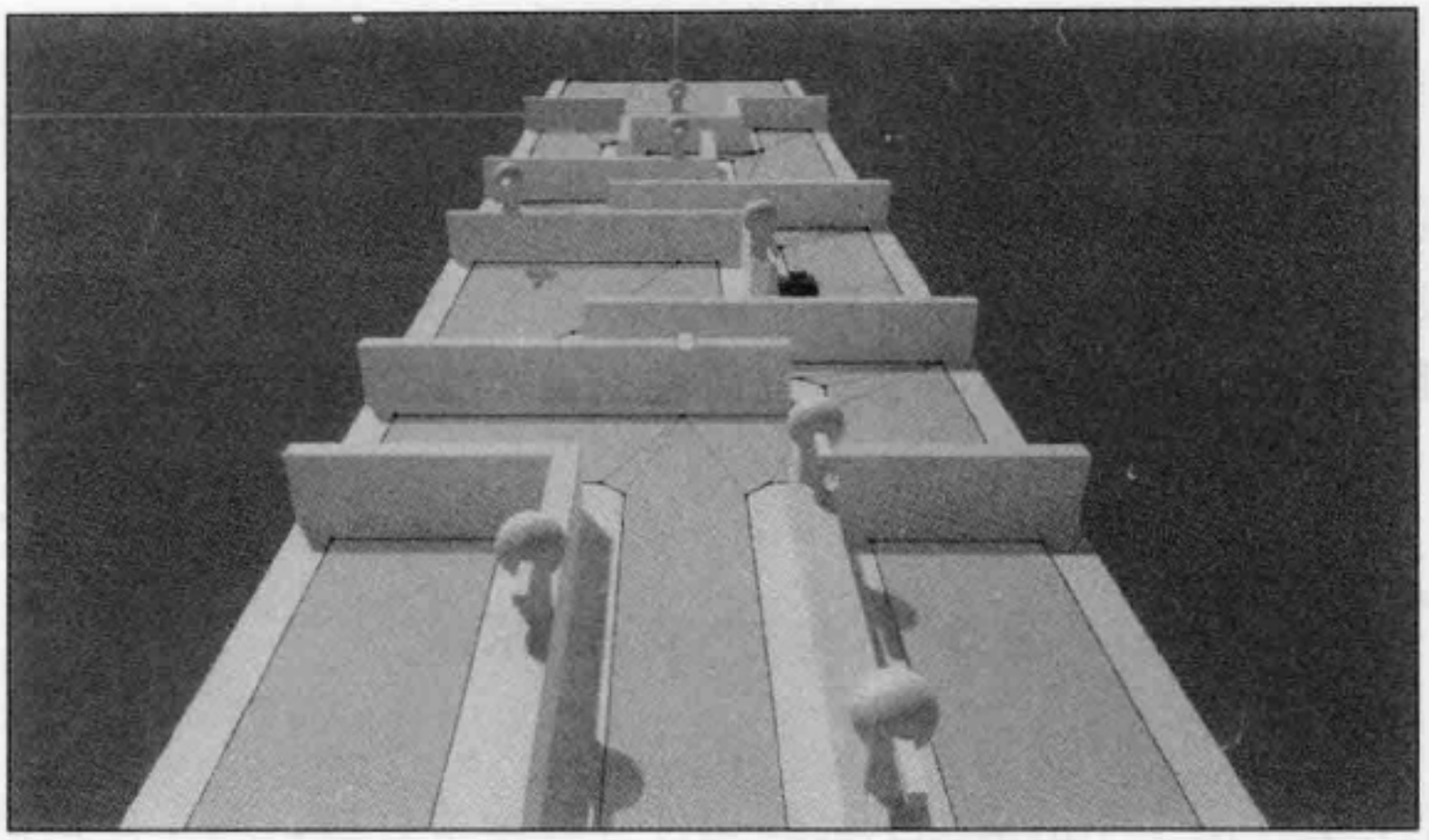

图 8.10

读者可根据个人喜好对墙体和高塔重新布局。回忆一下，添加至场景中的障碍物对象应标记为静态，待全部操作设置完毕后，还需要重新烘焙场景导航网格。

8.5　测 试 示 例

用户可单击 Play 按钮观察坦克的行进方式。需要注意的是，此处添加了画布对象，相关标记解释了玩家的操控方式。该过程并无太多新奇之处，如图 8.11 所示。

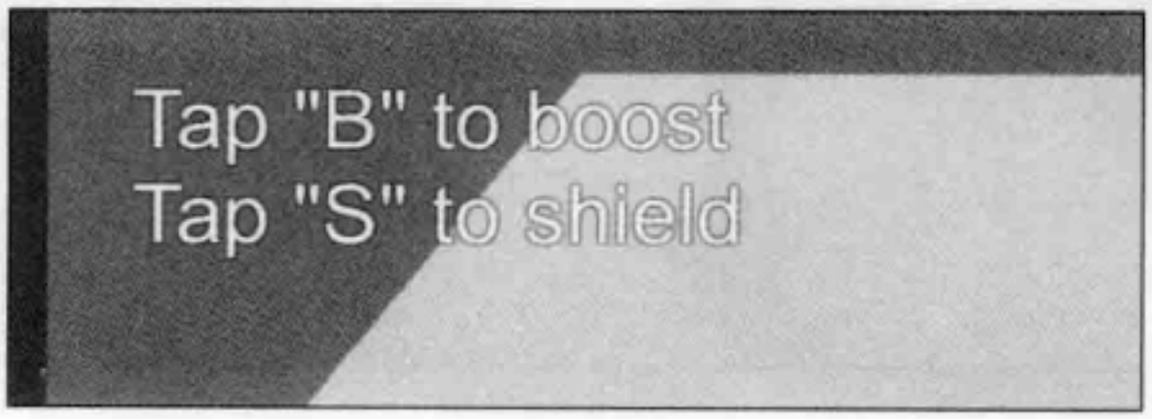

图 8.11

读者可对当前示例项目进行扩展，进而丰富游戏中的各项功能。例如，可扩展高

塔对象的类型、坦克对象的各种能力以及游戏规则，甚至可令坦克具备更加复杂、细微的行为。当前示例项目涵盖了状态机、导航机制、感知系统以及转向行为，全部内容整合至一个简单的示例中，对应结果如图 8.12 所示。

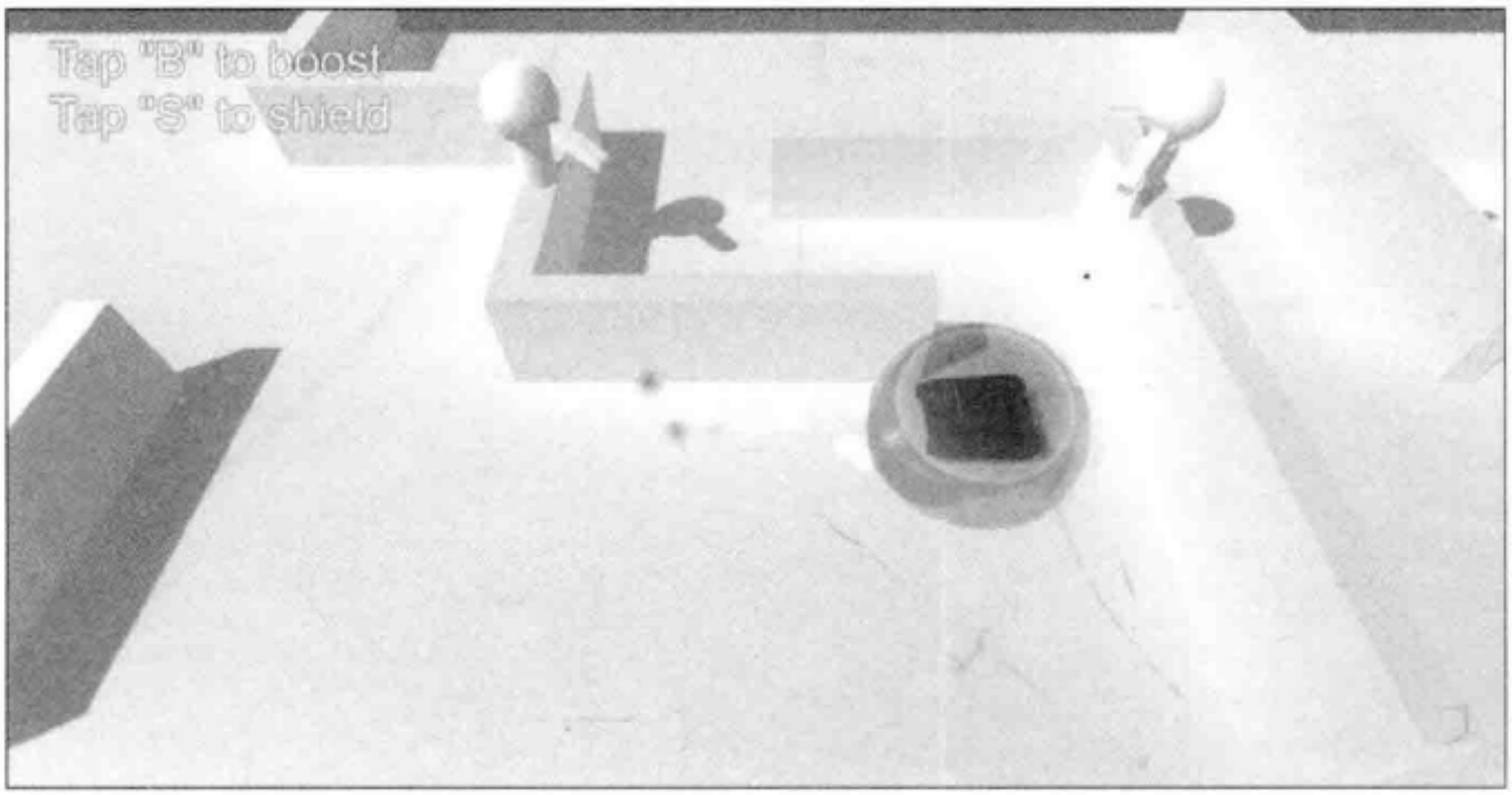

图 8.12

8.6 本 章 小 结

本章涉及书中的多个概念，并以此构建一款简单的坦克攻防游戏。其中包括第 2 章讨论的有限状态机，同时还创建了人工智能以驱动敌方高塔对象的行为。随后可将其与感知系统进行整合，进而提升游戏体验。最后，通过 Unity 的 NavMesh 特性实现了导航机制，引导坦克对象穿越迷宫关卡，经受敌方 AI 高塔对象的火力考验，其 AI 仅包含一个简单的目标，即摧毁敌方。